Titre: Title:	Recherche locale basée sur les contraintes pour la planification d'horaires de ligues sportives
Auteur: Author:	Marc Brisson
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Brisson, M. (2004). Recherche locale basée sur les contraintes pour la planification d'horaires de ligues sportives [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7347/

UNIVERSITÉ DE MONTRÉAL

RECHERCHE LOCALE BASÉE SUR LES CONTRAINTES POUR LA PLANIFICATION D'HORAIRES DE LIGUES SPORTIVES

MARC BRISSON
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2004

Your file Votre référence
ISBN: 0-494-01292-7
Our file Notre référence
ISBN: 0-494-01292-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

RECHERCHE LOCALE BASÉE SUR LES CONTRAINTES POUR LA PLANIFICATION D'HORAIRES DE LIGUES SPORTIVES

présenté par: BRISSON Marc

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BILODEAU Guillaume-Alexandre, Ph.D., président

M. PESANT Gilles, Ph.D., membre et directeur de recherche

M. GENDRON Bernard, Ph.D., membre et codirecteur de recherche

M. ROUSSEAU Louis-Martin, Ph.D., membre

REMERCIEMENTS

Je remercie d'abord mon directeur de recherche, M. Gilles Pesant, et mon codirecteur de recherche, M. Bernard Gendron. Ces deux personnes m'ont soutenu financièrement, mais surtout aidé et encouragé tout au long de ce projet de maîtrise. Je tiens à remercier spécialement M. Pesant pour son aide constante après son départ en sabbatique.

Je remercie aussi les collègues du Qu'ossé Ca (Québec Optimization and Satisfaction Strategies Explorating Constraint Algorithms) pour l'ambiance de travail.

Finalement, je remercie mes parents qui m'ont toujours encouragé à me dépasser.

RÉSUMÉ

L'optimisation combinatoire est une discipline bien connue pour laquelle plusieurs techniques de recherche de solutions ont été proposées. Dernièrement, plusieurs travaux de recherche dans ce domaine ont tenté de créer des modèles hybrides qui combinent les meilleures propriétés de différentes techniques. Pesant et Gendreau [26, 27] ont proposé une telle approche hybride basée sur la programmation par contraintes et la recherche locale. L'objectif de ce mémoire est de tester et tenter de raffiner ce modèle hybride. Pour cela, nous utilisons ce modèle pour optimiser deux problèmes de planification d'horaires de ligues sportives : le «Traveling Tournament Problem» (TTP) et le «Brazilian Soccer Scheduling Problem» (BSSP). Notre approche est une descente à voisinage variable basée sur cinq voisinages de recherche locale et un grand voisinage. Ces travaux nous ont menés à modifier la formulation des contraintes d'interface du modèle hybride de Pesant et Gendreau. Nous proposons d'exprimer ces contraintes sans les conditions de réveil. Cette nouvelle forme de contraintes a pour but d'utiliser plus activement les contraintes en incitant la recherche d'algorithme de filtrage pour des modifications plus élémentaires des domaines. Pour certaines contraintes d'interface, nous proposons un traitement qui est activé lors de modification aux bornes du domaine des variables impliquées. Nous avons découvert des classes de mouvements équivalents dans les voisinages. En combinant ces classes de mouvements équivalents à la fragmentation de domaine et, lorsque possible, au traitement activé sur les modifications des bornes des domaines des variables, nous obtenons les stratégies les plus efficaces d'exploration de voisinage. La descente sans le grand voisinage améliore les solutions du TTP d'environ 18% et environ 25% avec le grand voisinage. Pour le BSSP, l'amélioration sans grand voisinage est de 8.34% et monte à 10.49% avec. Malgré que nos résultats ne soient pas aussi bons que les meilleures techniques connues, la nouvelle formulation des contraintes d'interface est très prometteuse.

ABSTRACT

Combinatorial optimization is a well known discipline for which a great number of solution techniques have been proposed. Lately, many research works in that field have been concerned with creating hybrid models that combine the best properties of different basic techniques. One such models that combines constraint programming and local search was proposed by Pesant and Gendreau [26, 27]. The objective of this thesis is to test and refine the hybrid model proposed by Pesant and Gendreau. We have chosen to focus our efforts on two combinatorial optimization problems: the «Traveling Tournament Problem» (TTP) and the «Brazilian Soccer Scheduling Problem» (BSSP). The approach we propose combines constraint programming with a variable neighbourhood descent based on five local search neighbourhoods and one large neighbourhood. We also modify the formulation of the interface constraints in Pesant and Gendreau's model and express those constraints without wakeup conditions. This new formulation brings a more active use of the constraints by stimulating the search of the filtering algorithm for more elementary modifications of the domains. For some interface constraints, we also use a new process that is activated when the bounds of the constrained variables' domains are modified. Furthermore, we introduce classes of equivalent moves in the neighbourhoods, which combined with domain splitting and the activated process, gives us highly efficient neighbourhood exploration strategies. For instance, the variable neighbourhood descent without the large neighbourhood improves the TTP solutions by 18%, and improves the solutions by 25% when the large neighbourhood is used. For the BSSP, we observed an improvement of 8.34% without the large neighbourhood, and up to 10.49% with it. Although some of the solutions found by our approach are not as good as those obtained using the best known techniques, the new interface constraints fomulation showed great potential.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES NOTATIONS ET DES SYMBOLES

BSSP : Brazilian Soccer Scheduling Problem

CSP : Problème de satisfaction de contraintes (Constraint Satisfaction Problem)

DDS : Depth-bounded Discrepancy Search

GV : Grand voisinage

LDS : Recherche à déviations limitées (Limited Discrepancy Search)

LNS : Recherche à grand voisinage (Large Neighborhood Search)

NBL : Ligue nationale de baseball (National baseball league)

NSH : Voisinage $NewSwapHomes$

PNE : Programmation en nombres entiers

PPC : Programmation par contraintes

S2R : Voisinage $PartialSwapTwoRounds$

S2T : Voisinage $PartialSwapTwoTeams$

S3R : Voisinage $PartialSwapThreeRounds$

S3T : Voisinage $PartialSwapThreeTeams$

SH : Voisinage $SwapHomes$

TTP : Traveling Tournament Problem

VND : Descente à voisinage variable (Variable Neighborhood Descent)

VNS : Recherche à voisinage variable (Variable Neighborhood Search)

LISTE DES TABLEAUX

INTRODUCTION

Plusieurs entreprises et organismes de différents secteurs font face à des problèmes difficiles à résoudre. On peut penser tout simplement à la confection des horaires des employés. Les difficultés de la conception d'horaires sont de réduire les coûts, répondre à la charge de travail et respecter les conditions de travail. Le routage de véhicule est un autre bon exemple. Trouver les trajets les plus économiques pour des véhicules qui doivent visiter plusieurs clients n'est pas aisé. La conception de réseaux offrant les débits et la fiabilité voulus aux coûts les plus bas est aussi une activité critique. Ces problèmes génériques varient selon les besoins des entreprises et organismes. On peut trouver dans [6],[28] et [7] des exemples concrets pour les problèmes présentés.

Il existe ainsi un nombre important de problèmes ayant motivés plusieurs travaux de recherche. De façon générale, on distingue problèmes de satisfaction et problèmes d'optimisation. Les problèmes de satisfaction sont des problèmes qui ont beaucoup de contraintes et il est difficile de trouver une solution capable de toutes les satisfaire. L'objectif est donc de trouver une solution capable de satisfaire toutes les contraintes ou, si ce n'est pas possible, d'en maximiser le nombre de contraintes satisfaites. Les problèmes d'optimisation sont des problèmes de satisfaction pour lesquels c'est la meilleure solution valide qui est recherchée. Pour plusieurs de ces problèmes, c'est le coût qui doit être minimisé.

Il existe différentes techniques pour satisfaire et optimiser ces problèmes. Ces techniques proviennent généralement de trois domaines : la programmation mathématique, les méta-heuristiques et la programmation par contraintes. Les techniques ont des caractéristiques qui les distinguent les unes des autres et il est parfois intéressant de combiner certaines d'entre elles. Les méthodes hybrides qui en ré-

sultent peuvent alors bénéficier des avantages de chacune des techniques utilisées.

Objectifs

Pesant et Gendreau [26, 27] ont proposé une approche hybride basée sur la programmation par contraintes et la recherche locale. D'une part, la programmation par contraintes est utilisée pour l'expression, la gestion et la vérification des contraintes. D'autre part, la recherche locale est utilisée pour optimiser les problèmes. Cette technique s'est montrée efficace pour modéliser et explorer des grands voisinages de recherche locale. L'objectif de ce mémoire est de tester et tenter de raffiner ce modèle hybride. Pour cela, nous avons décidé de tester ce modèle sur des problèmes appropriés.

Le premier problème que nous avons retenu est le «Traveling Tournament Problem» (TTP). Ce problème, proposé par Easton, Nemhauser et Trick [8], a été conçu pour faire ressortir les difficultés fondamentales de la création d'horaires sportifs où le transport et les patrons de matchs à domicile et à l'extérieur sont importants. Ce problème est directement inspiré des travaux effectués pour la planification de la ligue nationale de baseball (NBL). Malgré les connaissances du domaine de la planification d'horaires pour la satisfaction des patrons et celles du problème de voyageur de commerce pour l'optimisation des distances, la combinaison de satisfaction et optimisation rend le TTP très difficile à résoudre même pour de petites tailles. Cela, autant pour les méthodes de recherche opérationnelle que celles de programmation par contraintes. Le TTP semble donc être un bon problème pour des approches hybrides. Il a donc été choisi pour cette raison, mais aussi parce qu'il semble un bon candidat pour des voisinages de grande taille. Le second problème que nous avons choisi est le «Brazilian Soccer Scheduling Problem». Ce problème représente la planification d'un horaire pour le championnat des 24 équipes de la

première division brésilienne de soccer. Sa structure est une spécialisation du TTP, il présente donc les mêmes caractéristiques qui nous ont fait choisir le TTP.

Nous allons donc appliquer le modèle hybride permettant la recherche locale en programmation par contraintes à ces deux problèmes, ce qui nous permettra de les optimiser. Pour cela, nous commencerons par modéliser ces deux problèmes en programmation par contraintes. Ces modèles permettront de satisfaire les problèmes de façon exacte et de trouver un certain nombre de solutions valides. La recherche locale utilisera ces solutions comme point de départ. La recherche locale sera en fait une descente à voisinage variable où l'utilisation de plusieurs voisinages permet à la recherche de quitter les optima locaux. Nous définirons ensuite les voisinages de recherche locale pour le TTP que nous réutiliserons pour le BSSP avec le moins de modifications possibles. Nous chercherons les meilleures stratégies d'exploration de ces voisinages.

La descente à voisinage inclura aussi une recherche à grand voisinage. Il servira lorsque tous les voisinages de recherche locale seront bloqués dans un optimum local.

Plan du mémoire

Le mémoire se divise en cinq chapitres. Le premier chapitre explique les fondements de la programmation par contraintes comme la propagation et la cohérence. Le second chapitre introduit les algorithmes de recherche locale comme : la simple recherche locale, la recherche à voisinage variable, la recherche tabou et le recuit simulé. Ce chapitre présente aussi une revue de littérature des méthodes hybrides combinant la programmation par contraintes et la recherche locale. Le troisième chapitre est une description des deux problèmes que nous avons choisis ainsi qu'une

revue des travaux antérieurs sur ces problèmes. Le quatrième chapitre présente la mise en oeuvre de ce projet : les modèles de programmation par contraintes, la descente à voisinage variable, les voisinages de recherche locale et le grand voisinage. Le dernier chapitre présente tous les résultats obtenus ainsi que les stratégies d'exploration des voisinages.

CHAPITRE 1

PROGRAMMATION PAR CONTRAINTES

La programmation par contraintes (PPC) est l'étude des systèmes automatisés de calcul basés sur les contraintes. Ce paradigme de haut niveau utilise les contraintes pour exprimer et résoudre des problèmes de grande taille, particulièrement des problèmes combinatoires. L'idée de base est d'exprimer les problèmes de façon naturelle avec des variables et des contraintes pour ensuite les résoudre à l'aide d'un *solveur* de contraintes basé sur la *propagation* de contraintes et le *filtrage* des domaines (notions qui seront introduites à la section 1.3). Le solveur est conçu pour résoudre tous les problèmes qui peuvent être exprimés à l'aide d'un certain *domaine de contraintes* (section 1.1). Ainsi, pour pouvoir résoudre un problème à l'aide du solveur, il suffit de procéder à sa modélisation, c'est-à-dire son expression sous forme de contraintes pouvant être traitées par le solveur. Cette séparation entre l'expression et la résolution du problème est un des principaux avantages de la programmation par contraintes. Des problèmes très complexes peuvent ainsi être exprimés facilement et résolus de façon efficace.

1.1 Domaine de contraintes

Deux éléments déterminent le domaine de contraintes : d'abord D, le domaine des valeurs que peuvent prendre les variables, et ensuite, la signature Σ, qui est l'ensemble des symboles de fonctions et prédicats disponibles. Les contraintes sont des relations exprimées par des formules closes formées de variables et de symboles de Σ. Ainsi, un domaine de contraintes linéaires sur les réels aura une signature

$\Sigma = \{+, =, <, \leq, 0, 1\}$ et le domaine des variables sera l'ensemble des nombres réels $D = \Re$. La PPC offre plus de liberté dans l'expression des contraintes que la programmation mathématique en nombres entiers (PNE). Les contraintes d'inégalité, les contraintes d'implication et l'indexation de vecteurs à l'aide de variables, par exemple, sont faciles à utiliser. La programmation par contraintes s'est avérée être particulièrement efficace pour les contraintes sur domaines finis, c'est-à-dire lorsque le domaine de chacune des variables est limité à un ensemble fini de valeurs. La description plus détaillée de la PPC qui suit concerne donc les contraintes sur domaines finis.

1.2 Modélisation

Un problème de PPC est exprimé comme un problème de satisfaction de contraintes (CSP). Un ensemble de variables $X = \{x_1, x_2, ..., x_n\}$ sert à représenter le choix de la solution. Chacune de ces variables admet différentes valeurs possibles provenant d'un domaine fini $\{D(x_1), D(x_2), ..., D(x_n)\}$. La modélisation du problème se complète par l'ensemble des contraintes $C = \{c_1, c_2, ..., c_m\}$ auxquelles doivent être soumises les variables. Une solution à un tel modèle est un ensemble de valeurs $V = \{v_1, v_2, ..., v_n\}$ tel que lorsque $X = V$ $(x_1 = v_1, x_2 = v_2, ..., x_n = v_n)$ les contraintes de C sont toutes satisfaites. Une telle affectation est dite *cohérente*. L'*espace de recherche* de solutions est l'ensemble de toutes les affectations V (cohérentes et non cohérentes) et se définit par le produit cartésien $D(x_1) \times D(x_2) \times ... \times D(x_n)$ des domaines des variables.

La programmation par contraintes peut aussi être utilisée pour des problèmes d'optimisation. L'optimisation consiste à trouver la meilleure solution possible à un problème. La qualité des solutions est évaluée par une fonction objectif que l'on doit minimiser ou maximiser selon la nature du problème. Un problème d'optimisation

est modélisé en programmation par contraintes comme un problème de satisfaction de contraintes auquel la valeur de la fonction objectif est ajoutée. Pour cela, il faut ajouter au modèle une variable *obj* qui représente la valeur de la solution. Typiquement, pour chaque solution trouvée, une contrainte est ajoutée sur la variable *obj* pour que les futures solutions trouvées soient meilleures.

Lorsque l'espace de recherche est très grand, trouver les affectations valides ne peut se faire en énumérant toutes les affectations. En PPC, le solveur utilise les contraintes pour réduire l'espace de recherche.

1.3 Solveur : propagation et filtrage

Alors que les variables du modèle de PPC servent à représenter le choix d'une solution, les contraintes servent à traduire les exigences du problème. Elles restreignent donc les domaines pour obtenir une affectation cohérente. Si, par exemple, deux variables x et y, ayant comme domaines $D(x) = \{0, 1, 2, 3\}$ et $D(y) = \{1, 2, 3\}$, sont soumises à la contrainte $x < y$, alors la valeur 3 du domaine de x n'est plus possible car il n'y a aucune valeur plus grande dans le domaine de y. On appelle *filtrage* ce processus qui enlève des domaines des variables les valeurs qui ne sont pas valides. De plus, si la variable x est impliquée dans une deuxième contrainte $z = x + y$, où $D(z) = \{1, 2, 3, 4, 5, 6\}$, alors l'élimination de la valeur 3 du domaine de x fait en sorte que z ne pourra jamais prendre la valeur 6. Le filtrage de x s'est donc propagé à la variable z. La *propagation* est le mode de communication des contraintes par l'entremise des domaines des variables. Ce sont ces deux concepts qui permettent à la PPC de réduire l'espace de recherche et qui la rendent ainsi efficace. Il existe quatre caractérisations générales du niveau de cohérence : la cohérence de noeud pour les contraintes unaires, la cohérence d'arc pour les contraintes binaires, la cohérence de bornes pour les contraintes arithmétiques et la cohérence de domaine

pour les contraintes impliquant plus de deux variables.

1.3.1 Cohérence de noeud

La cohérence de noeud est le niveau le plus simple. Si le domaine $D(x)$ d'une variable x contient une valeur v qui ne satisfait pas une des contraintes unaires appliquées sur x, alors la valeur v mènera toujours à une affectation non cohérente. La valeur v peut donc être éliminée de $D(x)$. Pour que la cohérence de noeud soit respectée, il faut donc éliminer les valeurs non cohérentes pour toutes les contraintes unaires. Par exemple, une variable x ayant comme domaine $D(x) = \{1, 2, 3, 4, 5, 6\}$ soumise à une contrainte $x > 3$ verra son domaine réduit à $D(x) = \{4, 5, 6\}$.

1.3.2 Cohérence d'arc

La cohérence d'arc concerne les contraintes liant deux variables. Le concept de base est de vérifier, pour chaque valeur v_i du domaine d'une des variables, s'il existe une valeur *support* v dans le domaine de l'autre variable permettant de satisfaire la contrainte. Toutes les valeurs v_i qui n'auront pas de support seront éliminées. Il est important de noter que le processus est directionnel : que le domaine de la première variable soit cohérent au sens des arcs avec celui de la seconde n'implique pas l'inverse. En reprenant l'exemple de la contrainte $x < y$ présenté au début de la section 1.3, on voit que $D(y)$ est cohérent au sens des arcs puisque la valeur $0 \in D(x)$ est plus petite que toutes les valeurs de $D(y)$: la contrainte est ainsi toujours satisfaite. Cependant, $D(x)$ n'est pas cohérent au sens des arcs car pour la valeur $3 \in D(x)$, il n'y a aucune valeur dans le domaine de y qui soit plus grande. Cette valeur doit donc être filtrée de $D(x)$.

Contrairement à la cohérence de noeud, la cohérence d'arc ne peut être appliquée

une seule fois. Lorsque le domaine d'une variable impliquée dans une contrainte change, la cohérence de l'autre variable n'est plus assurée. Prenons les domaines filtrés des variables de l'exemple précédent $D(x) = \{0, 1, 2\}$ et $D(y) = \{1, 2, 3\}$ et enlevons la valeur 0 de $D(x)$. Le domaine $D(x)$ est toujours cohérent au sens des arcs puisque le domaine $D(y)$ n'a pas changé. Le domaine $D(y)$ doit cependant être vérifié puisque $D(x)$ a changé : le support d'une valeur de $D(y)$ dans $D(x)$ peut avoir été enlevé. C'est effectivement le cas, $1 \in D(y)$ doit être filtré car son seul support $0 \in D(x)$ a disparu. Puisque $D(y)$ vient d'être modifié, il faut vérifier $D(x)$ et constater que la cohérence est respectée.

1.3.3 Cohérence de bornes

La cohérence de bornes est utilisée pour les contraintes arithmétiques ($\Sigma \subset \{+, -, \times, \div\}$). Le principe est de ne plus travailler sur le domaine complet des variables mais plutôt sur leurs bornes. Ainsi, pour chaque variable on cherchera la plus grande et la plus petite valeur menant à une solution en relaxant la cohérence sur les réels. Pour chaque variable x, il faut que pour la plus petite valeur dans le domaine $min(D(x))$, il existe une valeur réelle r_i pour chacune des autres variables x_i avec $min(D(x_i)) \leq r_i \leq max(D(x_i))$ telle que l'affectation de la variable x à $min(D(x))$ et de toutes les autres variables x_i à r_i satisfasse la contrainte. Les exigences sont les mêmes pour la borne supérieure du domaine.

Par exemple, soit une contrainte $x = y + z$; l'étude des minimums et maximums mène aux règles de propagation suivante :

$$\begin{aligned} x &\geq min(D(y)) + min(D(z)), & x &\leq max(D(y)) + max(D(z)), \\ y &\geq min(D(x)) - max(D(z)), & y &\leq max(D(x)) - min(D(z)), \\ z &\geq min(D(x)) - max(D(y)), & z &\leq max(D(x)) - min(D(y)). \end{aligned}$$

1.3.4 Cohérence de domaine

La cohérence de domaine est une généralisation de la cohérence d'arc aux contraintes ayant plus de deux variables. Elle permet ainsi de maintenir les domaines cohérents pour toutes les contraintes du modèle. Malheureusement, il n'existe pas d'algorithme générique de filtrage efficace pour la cohérence de domaine. Généralement, la complexité des algorithmes de filtrage est exponentielle avec le nombre de variables impliquées dans la contrainte. Pour cette raison, il est difficile d'utiliser globalement la cohérence de domaine dans un solveur de programmation par contraintes. Il faudra alors recourir à la cohérence de bornes lorsque les contraintes sont de nature arithmétique ou à une transformation de la contrainte en plusieurs contraintes binaires et utiliser la cohérence d'arc.

Pour certaines contraintes globales (contraintes impliquant plusieurs variables), il est quand même avantageux de chercher à utiliser la cohérence de domaine. D'abord, parce que la transformation en contraintes binaires n'est pas toujours facile. Ensuite, parce que les contraintes globales oeuvrent sur les domaines de toutes les variables impliquées en même temps; elles possèdent donc beaucoup plus d'information pour effectuer le filtrage qu'une transformation en contraintes binaires. Les contraintes globales doivent posséder un algorithme de filtrage efficace pour que leur utilisation soit rentable. Ces algorithmes sont généralement complexes et proviennent souvent de domaines de recherche connexes comme la recherche opérationnelle ou la théorie des graphes et réseaux. Par exemple, dans plusieurs problèmes, il faut s'assurer qu'un ensemble de variables soit affecté à des valeurs toutes différentes. Ainsi, pour s'assurer que les trois variables x, y et z soient toutes différentes, on poserait les contraintes binaires $x \neq y \wedge x \neq z \wedge y \neq z$. Cependant, les domaines $D(x) = \{0, 1\}$, $D(y) = \{0, 1\}$ et $D(z) = \{0, 1\}$, qui sont cohérents au sens des arcs, ne mènent à aucune solution valide. La cohérence d'arc

$$X = \{0, 2, 2, 2\}$$
$$X = \{2, 2, 2, 0\}$$
$$X = \{0, 1, 1, 0\}$$
$$X = \{1, 1, 1, 0\}$$

Figure 1.1: Affectations valides pour un exemple de *Stretch*

ne suffit pas pour cette contrainte. La contrainte globale $AllDiff(x_1, x_2, ..., x_n)$, qui correspond à $x_1 \neq x_2 \neq ... \neq x_n$, a donc été conçue pour obtenir une meilleure puissance de filtrage. L'algorithme de filtrage de cette contrainte provenant de notions de la théorie des graphes est basé sur la recherche d'un couplage dans un graphe biparti. Le filtrage est beaucoup plus efficace et la cohérence de domaine est obtenue. van Hoeve [16] présente une revue chronologique des développements de cette contrainte.

La contrainte globale $Stretch(X, SeqMax, SeqMin)$, présentée par Pesant [25], est aussi très utile. Elle sert à limiter les longueurs des séquences de variables affectées à la même valeur dans un vecteur de variables. Soit X ce vecteur de n variables; l'union des domaines des n variables de X donne un ensemble de valeurs $V = \{D(x_1) \cup D(x_2) \cup ... \cup D(x_n)\}$. Les vecteurs $SeqMin = \langle SeqMin_{v_1}, SeqMin_{v_2}, ..., SeqMin_{v_m} \rangle$ et $SeqMax = \langle SeqMax_{v_1}, SeqMax_{v_2}, ..., SeqMax_{v_m} \rangle$ sont les limites de longueur des séquences pour les m valeurs de V. Ainsi, $SeqMin_{v_1}$ et $SeqMax_{v_1}$ sont les limites de la première valeur de V. Il est important de noter que ces limites sont effectives uniquement si les valeurs sont présentes dans le vecteur. Par exemple, si X est un vecteur de quatre variables ayant pour domaines $\langle D(x_1) = \{0, 1, 2\}$, $D(x_2) = \{0, 1, 2\}$, $D(x_3) = \{1, 2\}$, $D(x_4) = \{0, 2\} \rangle$ alors $V = \{0, 1, 2\}$. Posons les limites $SeqMin = \langle 1, 2, 3 \rangle$ et $SeqMax = \langle 2, 3, 3 \rangle$; alors la figure 1.1 présente toutes les affectations valides.

En programmation par contraintes, les algortihmes de filtrages sont appelés très

souvent et les domaines changent peu entre les appels. Il est donc très utile de rendre ces algorithmes incrémentiels; c'est-à-dire pouvoir récupérer le traitement fait précedemment et ne travailler que sur les changements des domaines. De cette façon, les algorithmes sont plus efficaces.

1.3.5 Propagation

Chaque contrainte possède donc un algorithme de filtrage lui permettant de maintenir les domaines de ses variables à un certain niveau de cohérence. Certaines variables sont impliquées dans plus d'une contrainte. Lorsque le domaine d'une de ces variables est réduit par une contrainte, la cohérence de son domaine n'est plus assurée pour les autres contraintes. Il faut donc les «réveiller» et appliquer leur algorithme de filtrage. Cette réaction en chaîne de réveil de contrainte et de transfert d'information au travers des domaines est la *propagation.*

De façon générale, les contraintes sont réveillées lorsque le domaine d'une de leurs variables est modifié (*WhenDomain*), c'est-à-dire lorsqu'une ou plusieurs valeurs ont été enlevées du domaine de cette variable. On distingue deux cas plus spécifiques de modification de domaine : lorsqu'une des bornes du domaine est modifiée (*WhenRange*) et lorsque le domaine est réduit à une seule variable (*WhenValue*). Le choix de l'événement de réveil dépend principalement de l'algorithme de filtrage de la contrainte. L'objectif est de réveiller la contrainte pour des modifications au domaine menant à un bon filtrage. Par exemple, la contrainte d'inégalité $x \neq y$ est réveillée sur les événements *WhenValue* puisqu'il n'y a pas de filtrage tant que le domaine de x ou y contient au moins deux valeurs. Les contraintes arithmétiques utilisant la cohérence de bornes sont plutôt réveillées sur les événements *WhenRange.* Dans certains cas, il est préférable de ne pas réveiller trop souvent les contraintes ayant des algorithmes de filtrage moins rapides, car les effets positifs du

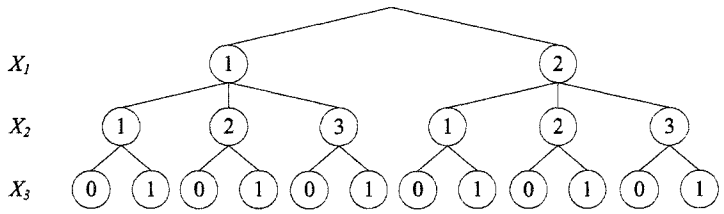

Figure 1.2: Fouille arborescente

filtrage sont annulés par le temps de filtrage. Il faut donc doser la propagation pour que l'exploration de l'arbre de recherche élagué par la propagation et le filtrage soit plus rapide que l'exploration de l'arbre complet.

1.4 Recherche de solutions

Il existe différentes méthodes pour résoudre des problèmes de satisfaction de contraintes. La méthode la plus simple consiste à générer les combinaisons de l'espace de recherche et vérifier pour chacune d'entre elles si les contraintes sont satisfaites. La fouille arborescente, figure 1.2, est une des méthodes les plus simples pour générer les combinaisons. À chaque niveau de l'arbre, une variable est affectée à une valeur de son domaine et une affectation de toutes les variables est atteinte aux feuilles de l'arbre. Les contraintes sont ainsi vérifiées aux feuilles et la fouille continue avec des retours-arrière tant qu'une affectation valide n'est pas trouvée.

La recherche de solutions en programmation par contraintes est aussi une fouille arborescente, mais l'utilisation des contraintes est beaucoup plus active. La propagation de contraintes permet de réduire considérablement l'espace de recherche pour trouver beaucoup plus rapidement les solutions valides. Il faut choisir la stratégie de recherche : l'ordre dans lequel les variables seront affectées et l'ordre

dans lequel les valeurs seront choisies. De façon générale, les variables sont choisies selon leur ordre d'apparition et les valeurs sont choisies en ordre lexicographique. Cependant, pour les problèmes complexes de grande taille, l'arbre de recherche réduit peut être encore très grand : chercher la solution optimale ou une solution valide devient alors beaucoup plus difficile. La stratégie de recherche doit alors être adaptée selon le problème. Les sections qui suivent présentent de quelle façon les stratégies de choix de variables et de valeurs peuvent être adaptées, en plus de décrire une technique appelée fragmentation de domaine et des approches qui n'explorent pas entièrement l'arbre de recherche.

1.4.1 Ordre de sélection des variables

L'ordre dans lequel les variables sont affectées est très important, car il influence la topologie de l'arbre de recherche. Selon la taille du domaine de la variable choisie, un noeud a plus ou moins de branches. Choisir en premier les variables de grand domaine génère ainsi un arbre très étalé à la racine. L'objectif en programmation par contraintes est de savoir le plus rapidement possible si un sous-arbre ne mène à aucune solution valide afin de l'élaguer. Les heuristiques de sélection de variables sont conçues pour chercher à satisfaire d'abord les portions les plus difficiles de l'arbre, celles où l'échec est le plus probable. L'ordre de sélection des variables peut être défini statiquement ou dynamiquement. Un ordre statique a l'avantage d'être très rapide car il est déterminé avant la recherche. L'ordre dynamique demande un traitement à chaque noeud pour choisir la variable à affecter, mais il peut exploiter l'information de la fouille pour prendre une meilleure décision. Pour cette raison, un ordre dynamique est normalement préféré. Voici quelques exemples :

- Plus petit domaine d'abord :

  On suppose que les variables ayant de petits domaines sont plus difficiles à

satisfaire, car elles ont peu de valeurs possibles.

- Plus petit domaine d'abord et plus grand nombre de contraintes :
  Le nombre de contraintes posées sur les variables sert de bris d'égalité lorsque des domaines sont de même taille.

- Moindre regret :
  Dans un problème d'optimisation, les valeurs affectées aux variables ont un impact sur la qualité de la solution. On peut ainsi évaluer l'impact de chacune des valeurs du domaine d'une variable. L'heuristique de moindre regret consiste à affecter en premier la variable qui présente le plus grand écart entre les deux meilleures valeurs du domaine. On appelle *regret* l'écart entre les deux meilleures valeurs d'un domaine. L'idée est de choisir d'abord la variable que l'on regretterait le plus de ne pas pouvoir choisir, car sa meilleure valeur est bien supérieure à la seconde meilleure valeur dans son domaine.

1.4.2 Ordre de sélection des valeurs

L'ordre de sélection des valeurs influence plus ou moins la recherche de solutions selon le type de problème. Lorsque l'arbre de recherche doit être complètement exploré, pour trouver toutes les solutions par exemple, l'ordre de sélection des valeurs n'est pas très important. Il ne détermine que l'ordre dans lequel les branches sont explorées et l'ordre dans lequel les solutions sont trouvées. Il prend de l'importance lorsqu'une seule solution est recherchée ou lorsqu'on optimise et que l'on cherche la solution optimale. Les heuristiques de sélection de valeurs sont conçues dans le but de mener le plus rapidement possible à la solution cherchée. Contrairement aux heuristiques de sélection de variables, ici on cherche d'abord à réussir. Idéalement, une heuristique dirigerait la recherche directement à une solution sans aucun retour-arrière. Il y a beaucoup de variétés d'heuristiques de choix de valeurs et elles

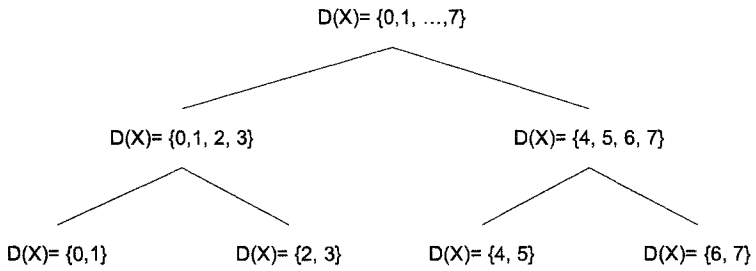

Figure 1.3: Fragmentation de domaine

utilisent généralement les caractéristiques propres au problème.

1.4.3 Fragmentation de domaine

La fragmentation de domaine est une technique qui modifie la stratégie de recherche. L'idée est de restreindre les domaines des variables pendant la recherche, mais sans nécessairement les fixer à une seule valeur à chaque noeud. La fragmentation suit le principe du moindre engagement : on ne veut pas restreindre trop rapidement le domaine. Ainsi, le branchement à un noeud ne se fait plus sur chaque valeur du domaine, mais plutôt sur un partitionnement plus grossier. On peut, comme le montre la figure 1.3, séparer le domaine en deux groupes. La fragmentation de domaine n'empêche pas l'utilisation d'heuristique de choix de valeurs : à un noeud, les valeurs peuvent être triées selon l'heuristique, les premières valeurs sont insérées dans le premier groupe et les autres dans le second. Cette technique est intéressante pour des variables ayant de grands domaines et lorsqu'il y a une bonne propagation malgré une moins grande réduction de domaine à chaque noeud.

1.4.4 Recherche partielle

L'arbre de recherche ne peut pas toujours être exploré complètement. Dans certains cas, le temps de recherche pour obtenir une solution est très limité, dans d'autres, la taille du problème est trop grande. Dans ce genre de situations, on peut recourir à une exploration partielle de l'arbre de recherche. Évidemment, lorsqu'il s'agit de problèmes d'optimisation, on ne pourra s'attendre à obtenir la solution optimale, mais plutôt des solutions de bonne qualité. Cette section présente quelques stratégies de recherche partielle provenant du survol fait par Focacci et al. [10].

La méthode la plus simple pour restreindre la recherche est de ne pas explorer toutes les branches des noeuds. L'exploration est plutôt réduite à une liste limitée de candidats pour chaque noeud. Ainsi, uniquement les branches les plus prometteuses selon l'heuristique de choix de valeurs sont explorées. Le nombre de candidats retenus peut être paramétré de façon à contrôler la taille de l'espace de recherche pouvant varier d'un candidat pour une fouille vorace à tous les candidats pour une fouille complète. L'heuristique peut être efficace lorsque le nombre de candidats est limité, mais trouver une bonne heuristique peut être très difficile. Ce sont normalement les premiers choix qui sont les moins bien guidés car la solution est mal définie. Pour aider l'heuristique dans cette situation, une méthode inspirée de la théorie des jeux consiste à explorer plus profondément l'arbre de recherche aux points de décision. Cette exploration permet de mieux évaluer l'impact de la branche choisie en donnant plus d'information à l'heuristique.

Une autre approche, la recherche à déviations, suit l'idée d'explorer l'arbre en respectant toujours l'heuristique de choix de valeurs, sauf pour un certain nombre de points de décision. Le principe sous-jacent est qu'il est plus vraisemblable que les bonnes solutions soient créées en respectant presque toujours l'heuristique qu'en divergeant souvent d'elle. Ainsi, la recherche à déviations limitées («Limited Dis-

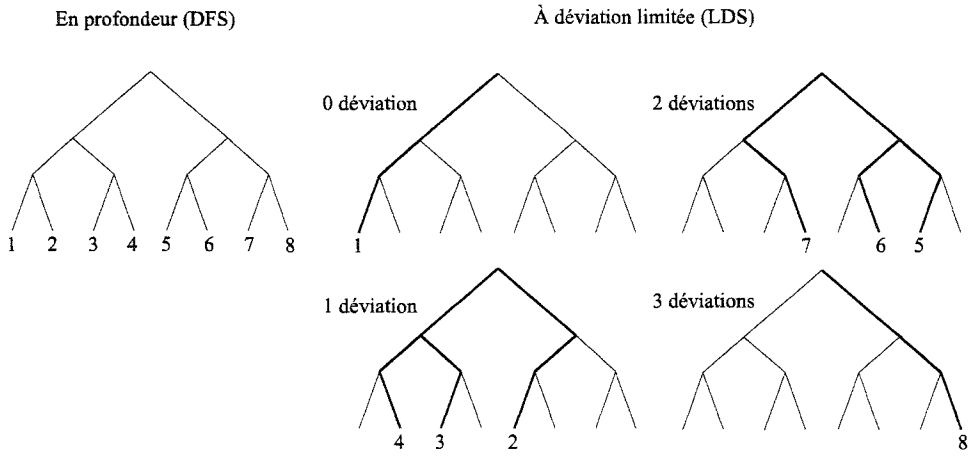

Figure 1.4: Recherche à déviations

crepancy Search» ou LDS) introduite par Harvey et Ginsberg [15] compte et limite le nombre de fois que l'heuristique n'est pas respectée. La figure 1.4 montre l'ordre dans lequel les feuilles sont parcourues selon une fouille en profondeur et selon des recherches à déviations limitées à 0, 1, 2 et 3 déviations. La branche de gauche correspond à suivre l'heuristique, alors que celle de droite y déroge. Pour la recherche à une déviation, on voit que tous les chemins explorés ne possèdent qu'une branche vers la droite. Le nombre de chemins explorés dépend donc du nombre de déviations permises. La recherche peut être élargie en effectuant successivement plusieurs LDS et en augmentant le nombre de déviations permises. La recherche à déviations pourrait être ainsi étendue à une recherche complète. La LDS est intéressante, car elle permet de diversifier rapidement les zones d'exploration, même pour les recherches très partielles. La DDS («Depth-bounded discrepancy search) proposée par Walsh [32] est une variante de la LDS. La DDS limite la profondeur à laquelle les déviations peuvent avoir lieu; passé cette profondeur, l'heuristique doit toujours être suivie.

CHAPITRE 2

ALGORITHMES DE RECHERCHE LOCALE

2.1 Recherche locale

Pour certains problèmes très difficiles, il n'existe pas d'algorithme exact assez efficace pour résoudre des exemplaires de grande taille; trouver la solution optimale ou simplement de bonnes solutions devient donc une entreprise ardue. Pour de tels problèmes, il est intéressant d'explorer l'avenue des heuristiques. Les heuristiques sont des algorithmes permettant de trouver des solutions plus efficacement que les algorithmes exacts, mais dont la qualité n'est pas garantie. Ainsi, une heuristique peut trouver des solutions quasi-optimales pour certains jeux de données et des solutions peu intéressantes pour d'autres.

La recherche locale est une heuristique d'amélioration simple et efficace pour l'optimisation de problèmes difficiles. Il s'agit d'une méthode itérative basée sur l'utilisation de voisinages. Soit une solution valide s du problème et une certaine façon de modifier légèrement cette solution, un *voisinage* V_s de la solution s est l'ensemble des solutions que l'on peut obtenir en modifiant s. On nomme *voisins* les éléments d'un voisinage et *mouvement* une modification menant à un voisin. Un exemple de voisinage simple consiste à essayer de permuter les valeurs de deux variables d'une solution; il y a donc autant de mouvements que de couples de variables à permuter. Chaque itération de la recherche locale est une exploration d'un voisinage pour une solution courante dans le but de trouver un voisin de meilleur coût que la solution courante. Si un tel voisin est trouvé, on le substitue à la solution courante pour l'itération suivante. Sinon, la recherche est terminée, car il n'y a plus de voisin de

Démarrer avec une solution initiale s_0
$i \leftarrow 0$
Tant que $\exists\ s \in V_{s_i}$ telle que $f(s) < f(s_i)$ **Faire**
  $i \leftarrow i + 1$
  $s_i \leftarrow s$
Fin Tant que

Figure 2.1: Schéma général de la recherche locale

meilleur coût. Soit f la fonction qui évalue le coût d'une solution ; la figure 2.1 présente le schéma général de la recherche locale pour un problème de minimisation.

On constate que la valeur de f diminue toujours, puisque seulement des voisins de meilleur coût sont acceptés. Le schéma de base de la recherche locale est donc une *descente* menant vers un optimum local. Cela signifie que dans le voisinage exploré, il n'existe pas de meilleure solution que celle atteinte. Cet optimum peut s'avérer être l'optimum global, mais aucune information ne permet de s'en assurer. Différentes métaheuristiques à voisinages ont été conçues dans le but de permettre à la recherche de quitter les optimums locaux; par exemple, la recherche tabou, le recuit simulé et la recherche à voisinage variable.

La recherche tabou présentée par Glover [11, 12, 13], consiste à choisir, à chaque itération le meilleur voisin même si celui-ci n'améliore pas la solution courante. Les mouvements ayant mené aux meilleurs voisins sont maintenus dans une liste de mouvements tabous (interdits) pour un certain nombre d'itérations. Cette liste sert à empêcher la recherche de revenir sur une solution déjà explorée, sinon la recherche pourrait être prise dans un cycle. La recherche peut ainsi quitter des optimums locaux, car des mouvements menant à des voisins moins bons que la solution courante sont acceptés.

L'exploration d'un voisinage selon le recuit simulé proposée par Kirkpatrick et al. [21] se fait aléatoirement, c'est-à-dire que les voisins sont choisis au hasard. Les voisins sont toujours acceptés et substitués à la solution courante lorsqu'ils l'améliorent. Dans le cas contraire, les voisins sont acceptés selon une probabilité qui décroît en fonction de l'avancement de la recherche et de l'écart à la solution courante. Cette métaheuristique est inspirée d'une procédure de métallurgie qui, pour produire des alliages de bonne structure cristalline, chauffe la matière pour ensuite la laisser refroidir lentement. Par analogie à cette procédure, c'est un paramètre T nommé *température* et Δf, la différence entre la valeur de la solution courante et le voisin considéré, qui contrôlent la probabilité d'acceptation des mauvais voisins. Comme le recuit est un refroidissement, la température diminue en cours de recherche, aidant à réduire ainsi la probabilité. Cette probabilité est souvent exprimée comme $p = e^{-\Delta f/T}$.

La recherche à voisinage variable introduite par Mladenovic et Hansen [24, 14] («Variable Neighborhood Search» ou VNS), contrairement à la recherche tabou et au recuit simulé, n'est pas basée sur un mécanisme d'acceptation de mauvais voisins pour quitter les optimums locaux. L'idée provient plutôt du fait que les optimums locaux d'un certain voisinage n'en sont probablement pas pour des voisinages de structure différente. Ainsi, c'est l'exploration de plusieurs voisinages qui permet à la recherche de quitter les optimums locaux. La recherche à voisinage variable peut être configurée pour effectuer seulement des descentes («Variable Neighborhood Descent» ou VND); voir figure 2.2. Les voisinages sont explorés à tour de rôle jusqu'à l'obtention d'un nouveau meilleur voisin. Cette nouvelle solution est substituée à la solution courante et la recherche recommence avec le premier voisinage. La descente se termine lorsque tous les voisinages sont bloqués. La VNS, figure 2.3, ajoute un aspect de diversification à la VND. À chaque itération, une solution s' choisie aléatoirement dans $V_i(s)$ sert de point de départ pour une méthode de

```
Définir l'ensemble des voisinages V_i, où i = 1, 2, ..., i_max
Trouver une solution initiale s
i ← 1
Tant que i ≤ i_max Faire
  Trouver le meilleur voisin s' ∈ V_i(s)
  Si f(s') < f(s) Alors
    i ← 1
    s ← s'
  Sinon
    i ← i + 1
  Fin Si
Fin Tant que
```

Figure 2.2: Descente à voisinage variable

recherche locale. Si s'', l'optimum local obtenu, est meilleur que s, alors s'' devient la nouvelle solution courante.

2.2 Recherche locale basée sur les contraintes

La programmation par contraintes et la recherche locale sont deux méthodes très différentes pour résoudre des problèmes d'optimisation combinatoire. La PPC exploite la propagation et le filtrage pour maintenir les domaines des variables à un certain niveau de cohérence. Cela lui permet d'élaguer l'arbre de recherche et d'accélérer la recherche de solutions. Elle ainsi capable de résoudre des problèmes très contraints et même de prouver l'irréalisabilité de certains. La recherche locale explore plutôt des voisinages de solutions existantes dans le but de les améliorer. Cette approche permet d'explorer de façon désordonnée l'espace de recherche et ainsi accéder plus rapidement à de bonnes solutions. Aucune garantie sur la qualité des solutions ne peut cependant être donnée et l'irréalisabilité ne peut pas être détectée. L'étude des caractéristiques de ces méthodes [20] montre une certaine

Définir l'ensemble des voisinages V_i, où $i = 1, 2, ..., i_{max}$
Trouver une solution initiale s
$i \leftarrow 1$
Tant que $i \leq i_{max}$ **Faire**
  Générer aléatoirement une solution $s' \in V_i(s)$
  Appliquer une méthode de recherche locale avec x' comme solution initiale; noter l'optimum local atteint s''
  Si $f(s'') < f(s)$ **Alors**
    $i \leftarrow 1$
    $s \leftarrow s''$
  Sinon
    $i \leftarrow i + 1$
  Fin Si
Fin Tant que

Figure 2.3: Recherche à voisinage variable

complémentarité. Plusieurs chercheurs ont donc conclu qu'il est raisonnable de penser que l'hybridation de ces deux méthodes peut être intéressante pour certains problèmes.

Dans cette section, plusieurs techniques d'hybridation sont présentées. Ces techniques n'utilisent pas toutes intégralement la PPC telle que présentée au chapitre 1. De façon générale, la PPC représente une recherche arborescente avec un système de gestion des contraintes pour maintien de cohérence. Grossièrement, on peut séparer les approches en deux catégories : celles où la recherche locale est utilisée pendant la PPC et celles où la PPC est utilisée pendant la recherche locale.

2.2.1 Recherche locale pendant la PPC

La première méthode consiste à utiliser la recherche locale aux feuilles d'une recherche arborescente. Ces feuilles sont des solutions valides puisque l'affectation

est complète et que toutes les contraintes sont satisfaites. La recherche arborescente sert ainsi à générer des solutions initiales valides et la recherche locale dans une deuxième phase les améliore. Il s'agit donc d'une utilisation classique de la recherche locale pour améliorer des solutions existantes.

La recherche locale peut aussi être utilisée à des noeuds qui ne sont pas des feuilles, dans le but d'améliorer ou souvent de réparer les solutions partielles. L'évaluation des solutions partielles est cependant beaucoup plus difficile que celle de solutions complètes. Pour obtenir une évaluation de qualité, il faut normalement définir une bonne borne inférieure (dans les problèmes de minimisation). Une idée simple de voisinage consiste à choisir un sous-ensemble des variables affectées et à lui définir un voisinage. Un meilleur voisin dans ce voisinage est choisi et la propagation réduit ensuite le domaine des autres variables. La recherche globale peut alors continuer avec cet état partiel amélioré.

Une autre méthode est celle de sonde locale [20]. Il s'agit de la combinaison d'une fouille arborescente de haut niveau avec un sondeur. Le sondeur est un algorithme capable de résoudre facilement certaines contraintes. Les contraintes du problème sont ainsi séparées en deux groupes selon qu'elles sont faciles ou non pour le sondeur. La recherche de solution est une fouille arborescente qui génère des sous-problèmes constitués uniquement de contraintes faciles. Pour chaque sous-problème, le sondeur doit trouver une affectation complète. Cette affectation est ensuite vérifiée pour les contraintes difficiles. Si elles sont satisfaites, alors la solution est valide et la recherche peut terminer ou continuer pour trouver une solution de meilleur coût. Sinon, une nouvelle contrainte facile est ajoutée au sous-problème afin de forcer le sondeur à trouver une affectation qui viole une contrainte difficile de moins. L'ajout de la contrainte mène ainsi la fouille à un nouveau noeud. Lorsqu'un cul-de-sac est atteint, c'est qu'il ne fallait pas ajouter une des contraintes faciles posées. La contrainte ayant échoué (retour-arrière) est donc enlevée pour être remplacée par

sa négation. Le sondeur de l'application proposée dans [20] est un algorithme de recuit simulé. Cette méthode permet d'utiliser différents algorithmes pour le sondeur. Des sondeurs de programmation linéaire et de programmation mixte en nombres entiers ont ainsi été créés par Ajili et El Sakkout [1].

2.2.2 PPC pendant la recherche locale

Ces méthodes ont pour but d'utiliser les fouilles systématiques et la cohérence lors de la recherche locale pour réduire le voisinage et choisir de bons voisins. Lorsque les voisinages sont très grands, trouver le meilleur voisin ne peut plus se faire simplement en énumérant tous les voisins. L'espace de recherche est si grand que chercher le meilleur voisin devient un problème d'optimisation lui-même. Deux méthodes utilisant une solution réalisable s et un modèle de programmation par contraintes permettent d'implémenter l'exploration de grands voisinages en programmation par contraintes.

La première méthode est la recherche à grand voisinage (Large Neighborhood Search ou LNS) proposée par Shaw [29]. Un sous-ensemble des variables du modèle de PPC est fixé à la solution courante s, les autres variables sont réinitialisées à leurs domaines initiaux et le modèle est ensuite utilisé pour trouver des nouvelles solutions. Avant la LNS telle que proposée par Shaw, Applegate et Cook [3] utilisaient une méthode à grand voisinage similaire, pour un problème de planification de n tâches sur m machines. L'affectation des tâches était préservée sur toutes les machines sauf une. Toutes les combinaisons possibles des n tâches sur la machine libérée sont explorées par une recherche arborescente. La portion de la solution s préservée permet de contrôler la taille du voisinage. Différentes techniques peuvent être utilisées pour améliorer l'efficacité de cette méthode comme : contrainte d'optimisation pour ne générer que des voisins améliorant strictement la fonction

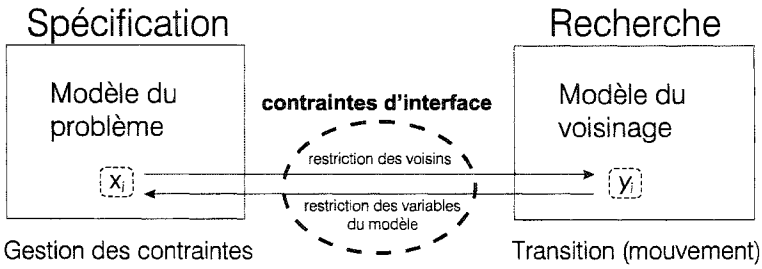

Figure 2.4: Recherche locale en PPC : Interaction entre le modèle du problème et le modèle du voisinage

objective, recherche à voisinage variable (Variable Neighborhood Search ou VNS) ou recherche à déviation limitée (LDS).

La deuxième méthode est celle proposée par Pesant et Gendreau [26, 27]. Ici, deux modèles de programmation par contraintes interagissent comme le montre la figure 2.4. Le modèle de gauche représente le problème combinatoire à optimiser. Le modèle de droite représente le voisinage de recherche locale. L'idée est de résoudre le second modèle et ainsi trouver les voisins valides de la solution courante. En cours de recherche, des contraintes d'interface traduisent le choix partiel ou complet du voisin vers le modèle du problème pour mener à la nouvelle solution partielle ou complète. Si les contraintes du modèle du problème sont violées, le voisin est alors rejeté. L'objectif est d'avoir des contraintes d'interface riches qui permettront d'élaguer rapidement l'arbre de recherche et ainsi trouver efficacement le meilleur voisin. Un avantage clair de cette méthode est la séparation entre la modélisation du problème et la modélisation de la recherche locale.

Soit un modèle de problème représenté par un ensemble de variables X et des contraintes; un simple voisinage consiste à échanger les valeurs de trois de ces variables. La figure 2.5 montre les valeurs de la solution courante s et les valeurs qui sont affectées à X après l'échange. L'échange peut être représenté par un

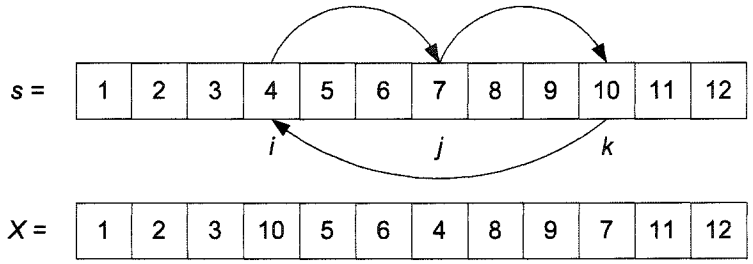

Figure 2.5: Recherche locale en PPC : Échange entre trois variables

modèle de voisinage de trois variables I, J et K déterminant les variables de X à permuter. Le domaine de ces variables est l'ensemble des indices de X et la contrainte $I < J < K$ doit être respectée pour éliminer des symétries. Explorer ce voisinage consiste à trouver toutes les solutions réalisables du modèle de voisinage. Pour traduire un échange (i,j,k) vers la nouvelle solution, il faut des contraintes d'interface:

$$(I = i) \Rightarrow \bigwedge_{\forall l<i} X_l = s_l \tag{2.1}$$

$$((I = i) \wedge (J = j)) \Rightarrow \bigwedge_{\forall l\ |i<l<j} X_l = s_l \tag{2.2}$$

$$((J = j) \wedge (K = k)) \Rightarrow \bigwedge_{\forall l\ |j<l<k} X_l = s_l \tag{2.3}$$

$$(K = k) \Rightarrow \bigwedge_{\forall l>k} X_l = s_l \tag{2.4}$$

Les contraintes d'interface (2.1) à (2.4) affectent aux variables qui ne sont pas impliquées dans l'échange leur valeur correspondante de la solution courante s. La contrainte d'interface (2.1) fixe toutes les variables précédant l'indice I, la contrainte d'interface (2.2) fixe toutes les variables entre les indices I et J, la contrainte d'interface (2.3) fixe toutes les variables entre les indices J et K et la contrainte

d'interface (2.4) fixe toutes les variables suivant l'indice K.

$$((I = i) \wedge (J = j)) \Rightarrow X_j = s_i \tag{2.5}$$

$$((J = j) \wedge (K = k)) \Rightarrow X_k = s_j \tag{2.6}$$

$$((K = k) \wedge (I = i)) \Rightarrow X_i = s_k \tag{2.7}$$

La contrainte d'interface (2.5) s'occupe de faire l'échange lorsque la variable I est affectée à la valeur i et que la variable J est affectée à la valeur j. On peut alors fixer la variable X_j à la valeur de la solution courante s_i. Les contraintes d'interface (2.6) et (2.7) font le même travail pour les autres variables.

Il est bon de séparer les contraintes d'interface afin de pouvoir les appliquer aussitôt que possible. Par exemple, si la contrainte d'interface (2.5) pour un couple (I=3,J=6) viole des contraintes du problème, alors toutes les combinaisons (3,6,k) sont éliminées. Si les contraintes d'interface (2.5), (2.6) et (2.7) avaient été remplacées par une seule contrainte d'interface appliquée uniquement lorsque les trois variables de voisinage sont fixées, alors toutes les combinaisons (3,6,k) auraient été testées.

Shaw et al. [30] proposent une méthode différente pour la recherche locale en PPC basée sur les *deltas*. Les *deltas* sont une structure de données permettant de représenter les voisins d'une solution. Ces structures préservent uniquement l'affectation des variables pouvant potentiellement être modifées dans la solution courante. L'utilisation des deltas se rapproche d'un sytème de recherche locale standard où, pour chaque voisin, il faut spécifier les changements aux valeurs des variables. Le mécanisme de deltas permet de structurer ce processus. La figure 2.6 est un exemple pour un voisinage qui inverse une seule variable d'un vecteur de booléens. Chaque feuille contient la variable qui est inversée ainsi que sa nouvelle valeur. L'exploration consiste à séparer récursivement en deux parties égales

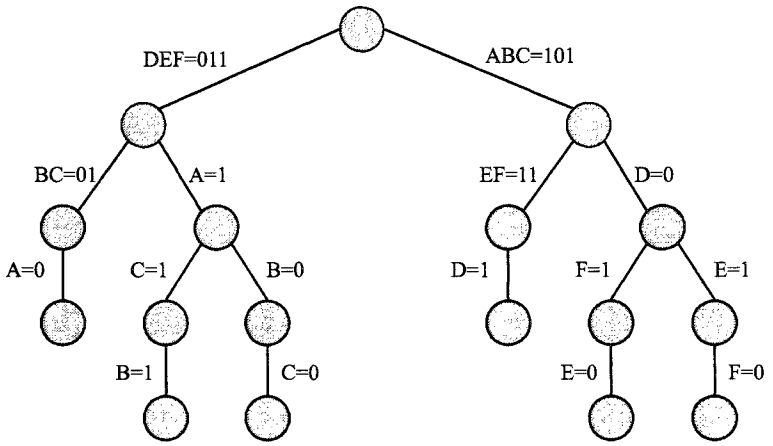

Figure 2.6: Deltas : Voisinage pour ABCDEF = 101011

l'ensemble des *deltas* de la solution courante. Les variables qui ne sont pas représentées dans la portion des *deltas* conservée sont affectées à leur valeur respective de la solution courante. Ainsi, la première branche vers la gauche de l'arbre consiste à garder les *deltas* modifiant les variables ABC, les variables DEF sont alors affectées à $D = 0$, $E = 1$, $F = 1$, comme indiqué sur l'arc. Shaw et al. proposent cette méthode comme une amélioration en terme de complexité sur celle de Pesant et Gendreau [27]. L'étude de complexité suppose que tous les voisins sont légaux; cependant, les deux méthodes profitent des contraintes pour élaguer de grandes portions des voisinages. La méthode de séparation des *deltas* peut donc mener à un problème d'équilibrage, car le nombre de voisins légaux dans chaque branche peut être très différent. Ainsi, expérimentalement, les deux méthodes montrent des résultats plutôt similaires en terme d'efficacité. Un inconvénient de cette approche est que les *deltas* doivent être énumérés avant l'exploration, ce qui n'est pas intéressant pour les grands voisinages. Pour pallier ce problème, les auteurs proposent de générer des groupes *deltas* et de les explorer indépendamment. La complexité de l'algorithme proposé n'est pas affecté tant que les groupes de variables sont assez grands pour que chaque variable soit mentionnée au moins une fois. Cette variante

est intéressante lorsque le premier voisin réalisable est recherché et non le meilleur.

La dernière méthode est celle présentée par Jussien et Lhomme [19]. Elle débute avec une solution partielle obtenue à la suite d'une série de décisions. Un algorithme de filtrage et de propagation est ensuite appliqué pour en déterminer la cohérence. Si la cohérence est valide, alors une décision de plus peut être faite. Sinon, c'est qu'il y a incompatibilité avec les décisions prises. Une phase de réparation identifie alors le conflit et ensuite s'en sert pour choisir un voisinage judicieux pour l'ensemble des décisions courantes; par exemple, l'inversion de la valeur d'une variable de décision booléenne.

2.2.3 Outils de programmation

L'algorithme de recherche locale basé sur les *deltas* proposé par Shaw et al. [30] est intégré dans le solveur de programmation par containtes ILOG Solver [18] depuis sa version 5 .

Localizer proposé par Michel et van Hentenryck [23] est un langage de modélisation permettant l'expression de haut niveau d'algorithmes de recherche locale. Les algorithmes écrits avec ce langage comportent trois composantes : des invariants, des voisinages et le contrôle. Les invariants sont utilisés pour introduire les dépendances fonctionnelles entre les structures de données. En plus des structures de données de base utilisées pour représenter une solution, des structures de données additionnelles sont utilisées pour préserver l'information redondante. Leurs valeurs sont spécifiées par des formules appelées invariants. Ces formules, qui sont similaires à des contraintes, sont utilisées pour maintenir des valeurs cohérentes dans les structures de données additionnelles à chaque fois qu'une valeur des structures de données de base est modifiée. *Localizer* est organisé autour des concepts tradition-

nels des algorithmes de recherche locale et propose des implémentations de plusieurs algorithmes de recherche locale comme l'amélioration locale, le recuit simulé et la recherche tabou. Des résultats intéressants sont obtenus, les performances étant comparable à celles d'algorithmes spécialisés.

Pour offrir un environnement de programmation d'algorithmes hybrides de PPC et recherche locale, Laburthe et Caseau [22] proposent de considérer séparément la logique et le contrôle et d'utiliser un langage spécifique pour contrôler la recherche de solutions. Pour cela, ils proposent le langage *SALSA* qui exprime les voisinages et les choix de recherche globale selon le même formalisme permettant ainsi l'expression d'algorithmes hybrides. Ce langage est dédié à la programmation d'algorithmes de recherche et doit être utilisé en coopération avec un langage de programmation hôte. *SALSA* augmente le niveau d'abstraction de la description d'algorithmes, cela permet d'améliorer le temps de développement, les coûts et la fiabilité du logiciel d'optimisation combinatoire.

COMET [31], proposé par Van Hentenryck et Michel, est un nouveau langage de programmation orientée objet conçu pour simplifier l'implémentation d'algorithmes de recherche locale. *COMET* propose une architecture basée sur les contraintes pour la recherche locale organisée autour de deux composantes principales: une composante déclarative, qui modélise l'application en terme de contraintes et de fonction, et une composante de recherche qui spécifie l'heuristique et la métaheuristique de recherche. Les contraintes sont des objets *différentiables* dans *COMET*; ils maintiennent un nombre de propriétés de façon incrémentielle et fournissent des algorithmes pour évaluer l'effet de différentes opérations sur ces propriétés. La composante de recherche utilise alors ces fonctionnalités pour guider la recherche locale à l'aide de sélecteurs multidimensionnels et autres structures de contrôle de haut niveau. L'architecture permet de faire des algorithmes modulaires de plus haut niveau. *COMET* sépare les composantes de modélisation et de recherche

permettant ainsi d'expérimenter différentes heuristiques de recherche sans affecter le modèle du problème. *COMET* a été utilisé pour plusieurs applications et, grâce à ses algorithmes incrémentiels, il peut être très compétitif face à des algorithmes spécialisés.

CHAPITRE 3

CONFECTION D'HORAIRES DE LIGUES SPORTIVES

Comme il a été présenté dans l'introduction, ce projet a pour but de tester et raffiner la technique de recherche locale basée sur les contraintes proposée par Pesant et Gendreau [26, 27] et présentée à la section 2.2.1. Pour faire cela, la technique sera adaptée à deux problèmes d'horaires sportifs qui serviront de banc d'essai.

La confection d'horaires de ligues sportives est désormais une activité très importante. Les contrats de télédiffusion, la vente de billets, la publicité et autres produits dérivés font du sport professionnel une industrie lucrative. Aux États-Unis, par exemple, le contrat pour la diffusion télévisuelle nationale et locale des matchs de baseball est de l'ordre de 800 millions de dollars [8]. Plusieurs ligues ont ainsi des revenus très élevés, mais doivent en retour s'assurer de livrer un bon produit. Pour cela, il est important de planifier aux heures de grande écoute les matchs réunissant les équipes les plus attrayantes. En plus de ces grandes ligues, il existe un nombre impressionnant de petites ligues sportives qui ont aussi besoin de planification. Ces ligues ont différentes contraintes qui rendent la création d'horaire difficile. Easton, Nemhauser et Trick [8] énumèrent plusieurs travaux de recherche qui ont ainsi été réalisés dans ce domaine. Ce chapitre présente les deux problèmes qui seront utilisées pour tester la méthode hybride.

3.1 Traveling Tournament Problem

La ligue nationale de baseball (*National Baseball League* ou NBL) est formée de 16 équipes réparties à travers tous les États-Unis, et même Montréal au Canada

Figure 3.1: TTP : Villes de la NBL

jusqu'en 2004 (voir les villes numérotées à la figure 3.1). Chacune de ces équipes joue 162 matchs durant une saison de 180 jours. Cette ligue cherche à minimiser les coûts de transport, mais cela doit se faire en respectant le fait que les équipes ne veulent pas voyager trop longtemps. Ces exigences font que la résolution d'horaire pour la NBL est trop difficile pour les méthodes d'optimisation actuelles. Pour cette raison, Easton, Nemhauser et Trick [8] ont proposé un problème qui est conçu pour faire ressortir les difficultés fondamentales de la création d'horaires sportifs où le transport et les patrons domicile/extérieur sont importants. Il s'agit du «Traveling Tournament Problem» (TTP) qui est directement inspiré de travaux effectués pour la NBL.

Malgré les connaissances du domaine de la planification d'horaire pour la satisfaction des patrons et celles du problème de voyageur de commerce pour l'optimisation des distances, la combinaison de satisfaction et d'optimisation rend le TTP très difficile à résoudre même pour de petites tailles, et cela, autant pour les méthodes de recherche opérationnelle que pour celles de programmation par contraintes. Le TTP semble donc être un bon problème pour des approches hybrides. Il a donc été

choisi pour cette raison, mais aussi parce qu'il semble un bon candidat pour des voisinages de grande taille. Nous présentons d'abord une description détaillée de ce problème, ainsi qu'une revue des précédents travaux.

3.1.1 Description

Au lieu de planifier une saison complète de la NBL, le TTP consiste à planifier un tournoi à la ronde («Round Robin») double pour n des 16 équipes de cette ligue. Un tournoi à la ronde est un horaire planifié en terme de rondes où chaque équipe doit jouer contre toutes les autres une seule fois. Le tournoi double du TTP consiste donc à planifier un horaire où chaque équipe joue deux fois contre toutes les autres équipes et cela une fois à domicile et une fois à l'extérieur. Il doit donc y avoir $2n$-2 rondes et chaque équipe joue exactement une fois par ronde. À cela, des contraintes simulant des exigences d'horaires sportifs sont ajoutées au problème. Ainsi, une équipe ne peut être plus de trois matchs consécutifs à domicile ou à l'extérieur et ne peut jouer deux matchs consécutifs contre le même adversaire. L'objectif est de trouver l'horaire minimisant la somme des distances de déplacement.

La figure 3.1 montre la disposition des villes de la NBL et la figure 3.2 est la légende qui sera utilisée pour représenter ces villes. Lorsque la taille du TTP est inférieure à 16, ce sont les n premières équipes de la légende qui sont utilisées. La figure 3.3 est un exemple d'horaire pour un TTP de huit équipes. Chaque colonne est une ronde du tournoi et chaque ligne représente l'horaire d'une équipe. Le symbole @ indique un match qui a lieu à l'extérieur. Pour l'équipe 0 (Atlanta), le match de la première ronde sera donc contre l'équipe 1 (New York) et aura lieu à New York.

#	Ville	#	Ville
0	Atlanta	8	St-Louis
1	New York	9	Milwaukee
2	Philadelphie	10	Houston
3	Montréal	11	Colorado
4	Floride	12	San Francisco
5	Pittsburgh	13	San Diego
6	Cincinnati	14	Los Angeles
7	Chicago	15	Arizona

Figure 3.2: TTP : Légende des équipes

T\R	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	@1	@2	@3	1	@4	@5	@6	2	3	4	@7	5	6	7
1	0	@3	@2	@0	2	@4	@5	@6	4	7	3	@7	5	6
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	@6	@5	6	@2	@7	@0	5	@1	@4	7	4
4	6	7	@5	@2	0	1	@7	5	@1	@0	@6	3	2	@3
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	3	7	@3	0	1	@5	@2	4	2	@0	@1
7	@5	@4	6	5	@6	2	4	3	@2	@1	0	1	@3	@0

Figure 3.3: TTP : Exemple d'horaire pour huit équipes

3.1.2 Travaux existants

Easton, Nemhauser et Trick [9] sont les premiers à s'être attaqués au TTP. Leur approche est un algorithme de «Branch and Price» . L'algorithme de «Branch and Price» est une méthode de séparation et évaluation progressive («Branch and Bound») avec relaxation linéaire. À chaque noeud de l'arbre de séparation et évaluation progressive, la relaxation linéaire est résolue par génération de colonnes. Pour calculer la valeur optimale de la relaxation linéaire, un sous-problème est résolu par programmation par contraintes. Cette méthode a permis d'obtenir les solutions optimales uniquement pour des problèmes de 4 et 6 équipes et montre que le problème est très difficile pour les tailles plus grandes. D'assez bonnes solutions ont été obtenues pour toutes les tailles du problème.

Benoist, Laburthe et Rottembourg [4] ont proposé une approche basée sur des relaxations lagrangiennes. Le problème est décomposé par équipe en sous-problèmes de voyageur de commerce contraint. La relaxation lagrangienne se fait au niveau des contraintes liant les équipes pour forcer les horaires des équipes à être synchronisés et ainsi produire un horaire global valide. L'architecture résultante est collaborative : un modèle de programmation par contraintes sert d'algorithme de recherche principal et la relaxation lagrangienne est encapsulée dans une contrainte globale qui contrôle les solveurs de sous-problèmes. Les résultats obtenus sont généralement moins bons que les meilleures solutions obtenues par Easton, Nemhauser et Trick [9], surtout pour les grandes tailles.

Les meilleurs résultats publiés à ce jour sont ceux obtenus par la méthode de recuit simulé d'Anagnostopoulos et al. [2]. Le grand voisinage exploré est formé de cinq mouvements complexes consistant à faire différentes permutations dans l'horaire : permuter le lieu de rencontre des deux matchs d'une équipe contre une autre, permuter complètement ou partiellement deux rondes et permuter complètement ou

partiellement l'horaire de deux équipes. Un élément important de cette méthode est que la contrainte limitant la longueur des séjours (à domicile ou à l'extérieur) à trois matchs, ainsi que la contrainte empêchant deux matchs consécutifs contre le même adversaire, sont exprimées comme des contraintes molles. L'idée est d'étendre le voisinage exploré en acceptant des solutions non réalisables violant ces contraintes au prix de pénalités à la fonction objectif. L'algorithme de recuit simulé inclut une stratégie d'oscillation stratégique. Cette stratégie consiste à faire varier le coût des pénalités durant la recherche pour équilibrer le temps d'exploration des solutions réalisables et non réalisables. Le coût des pénalités augmente avec le temps passé à explorer le domaine non réalisable alors qu'il est réduit lorsque c'est le domaine réalisable qui est exploré. Une technique de réchauffement est aussi utilisée pour aider à quitter les optimums locaux. Lorsque la recherche bloque (optimum local atteint), la température est augmentée à deux fois sa valeur au moment de la découverte de la meilleure solution. La recherche termine alors après un certain nombre de réchauffements non fructueux. Les meilleures solutions trouvées avec cette méthode sont obtenues après une longue exécution de plusieurs jours. Il a cependant été montré que de très bonnes solutions sont obtenues rapidement. Par exemple, l'algorithme prend environ 1000 secondes pour battre les anciennes meilleures solutions pour 12 équipes et ce schéma se répète pour toutes les tailles. La méthode est aussi robuste, car les pires solutions trouvées sont toujours meilleures que les anciennes meilleures solutions.

3.2 Brazilian Soccer Scheduling Problem

Le «Brazilian Soccer Scheduling Problem» (BSSP) [5] représente la planification d'un horaire pour le championnat des 24 équipes de la première division brésilienne de soccer. Il a été choisi parce que sa structure est une spécialisation du TTP.

3.2.1 Description

Le BSSP est un tournoi à la ronde double comme le TTP. Chaque équipe doit donc jouer exactement une fois par ronde et jouer deux fois (un match à domicile et un match à l'extérieur) contre toutes les autres équipes dans l'horaire. L'horaire doit cependant être conçu pour que la seconde moitié soit une répétition de la première où uniquement le lieu des matchs change. Ainsi, si les équipes A et B jouent l'une contre l'autre à la première ronde de l'horaire et que ce match se déroule au domicile de l'équipe B ($A@B$), alors le second match impliquant ces deux équipes doit avoir lieu au domicile de l'équipe A et cela à la première ronde de la seconde moitié de l'horaire. En plus de ces contraintes :

- Les équipes doivent jouer un match à domicile et un match à l'extérieur dans les deux premières rondes de chaque moitié de l'horaire.
- Les deux dernières rondes de chaque moitié doivent avoir la configuration inverse (en termes de domicile/extérieur) des deux premières rondes respectives.
- Les équipes d'un même état ne peuvent jouer entre elles à la dernière ronde (46 ème) de l'horaire
- La différence entre le nombre de matchs à domicile et le nombre de matchs à l'extérieur de chaque moitié doit être égal à un.
- Il ne doit pas y avoir plus de deux matchs consécutifs sur la route ou à domicile pour toutes les équipes.

La figure 3.4 présente la légende qui sera utilisée pour identifier les équipes. Cette figure montre aussi l'aéroport le plus près de chaque équipe ainsi que l'état dans

#	Équipe	Aéroport	État	#	Équipe	Aéroport	État
0	Atletico-MG	PLU	MG	12	Guarani	VCP	SP
1	Atletico-PR	CWB	PR	13	Internacional	POA	RS
2	Botafogo	SDU	RJ	14	Juventude	POA	RS
3	Corinthians	CGH	SP	15	Palmeiras	CGH	SP
4	Coritiba	CWB	PR	16	Paraná	CWB	PR
5	Criciuma	FLN	SC	17	Paysandu	BEL	PA
6	Cruzeiro	PLU	MG	18	Ponte-Preta	VCP	SP
7	Figueirense	FLN	SC	19	Santos	CGH	SP
8	Flamengo	SDU	RJ	20	Sao-Caetano	CGH	SP
9	Fluminense	SDU	RJ	21	Sao-Paulo	CGH	SP
10	Goiás	GYN	GO	22	Vasco	SDU	RJ
11	Grêmio	POA	RS	23	Vitória	SSA	BA

Figure 3.4: BSSP : Légende des équipes

lequel elles se trouvent. Les équipes sont situées dans différents états du Brésil et certaines équipes voisines partagent le même aéroport. L'objectif est de trouver l'horaire minimisant la somme des distances de déplacement plus l'écart entre l'équipe voyageant le plus et celle voyageant le moins. Les distances sont celles à vol d'oiseau entre les aéroports.

3.2.2 Travaux existants

À ce jour, la seule publication traitant le BSSP est celle de Biajoli et al. [5]. La technique proposée est directement inspirée du recuit simulé d'Anagnostopoulos et al. [2]. La méthode proposée ici n'explore cependant pas de solutions non réalisables et n'inclut pas de techniques d'oscillation stratégique ou de réchauffement. Le voisinage exploré est formé de trois mouvements. Le premier consiste à permuter le lieu de rencontre des deux matchs d'une équipe contre une autre. Le second permute deux matchs dans deux rondes d'une même moitié de l'horaire. Le dernier mouvement permute tous les matchs d'un couple d'équipes, échange A pour

B et vice versa dans tous les matchs impliquant ces deux équipes. Une procédure de raffinement par énumération limitée de solutions symétriques appliquée après le recuit simulé permet d'améliorer de 2% les solutions finales en quelques minutes. Étant la première méthode automatisée pour planifier ce tournoi, la meilleure solution à laquelle se compare cette méthode est celle générée manuellement par la Confédération brésilienne de soccer. Les résultats obtenus améliorent de 12,8% la fonction objectif et de 38,4% l'écart entre les équipes voyageant le plus et le moins.

CHAPITRE 4

MISE EN OEUVRE

Ce chapitre présente l'approche d'optimisation proposée pour la confection d'horaires de ligues sportives. Cette approche est une descente à voisinage variable explorant différents voisinages de recherche locale ainsi qu'un voisinage de grande taille similaire à ceux utilisés dans la recherche à grand voisinage. La première section de ce chapitre présente d'abord les modèles de programmation par contraintes utilisés pour la modélisation du TTP et du BSSP. La seconde section présente la descente à voisinage variable, c'est-à-dire les voisinages de recherche locale et la recherche à grand voisinage.

4.1 Modélisation du TTP

4.1.1 Variables

La modélisation du problème maître se fait à l'aide de trois tableaux M, A et L ayant $n \times (2n-2)$ variables. Les lignes sont les équipes et les colonnes les rondes. Soit i un indice pour les équipes (0 à n-1) et j un indice pour les rondes (0 à $2n$-3) du tournoi.

Les variables de décision sont les variables M. Chaque $M_{i,j}$ indique quel est l'adversaire de l'équipe i à la ronde j et à quel endroit aura lieu ce match. Comme il y a n adversaires possibles, mais que les matchs peuvent se jouer à domicile ou à l'extérieur, le domaine des variables $M_{i,j}$ contient $2n$ valeurs (0 à $2n-1$). La première moitié des valeurs représente les matchs contre les n équipes à l'extérieur

et la seconde moitié les matchs à domicile.

Les variables auxiliaires A et L représentent respectivement les adversaires et les lieux des matchs. Le domaine des variables A est l'ensemble des équipes (0 à $n-1$) alors que le domaine des variables $L_{i,j}$ ne contient que deux valeurs : 0 si le match de l'équipe i à la ronde j est sur la route et 1 s'il est à domicile. Ces variables ne servent pas à la prise de décision, mais plutôt à faciliter l'expression de différentes contraintes présentées plus loin. Pour lier les variables auxiliaires aux variables de matchs, deux vecteurs d'entiers sont utilisés :

$$ad = \langle 0, 1, ..., n-1, 0, 1, ..., n-1 \rangle$$

$$l = \langle \underbrace{0, 0, ..., 0}_{n}, \underbrace{1, 1, ..., 1}_{n} \rangle$$

Le vecteur ad est une double énumération des équipes alors que le vecteur l contient n fois la valeur 0 pour les matchs sur la route et n fois la valeur 1 pour les matchs à domicile. Les contraintes (4.1) et (4.2) qui suivent définissent les relations entre les variables M et les variables auxiliaires A et L.

$$A_{i,j} = ad[M_{i,j}] \tag{4.1}$$

$$L_{i,j} = l[M_{i,j}] \tag{4.2}$$

L'idée est d'indexer les vecteurs d'entiers avec les variables de match pour obtenir les valeurs des variables auxiliaires. Ainsi, pour un exemple de quatre équipes, les vecteurs d'entiers sont $ad = \langle 0, 1, 2, 3, 0, 1, 2, 3 \rangle$ et $l = \langle 0, 0, 0, 0, 1, 1, 1, 1 \rangle$. Un match $M_{i,j} = 3$ indique donc que l'adversaire de l'équipe i sera $ad[3] = 3$ et que ce match se disputera sur la route puisque $l[3] = 0$. Le match $M_{i,j} = 7$ est aussi un match contre l'équipe 3, mais celui-ci se disputera au domicile de l'équipe i puisque $l[7] = 1$. La figure 4.1 présente un exemple complet pour quatre équipes.

E\R	0	1	2	3	4	5
0	0	0	0	1	1	1
1	1	0	0	0	1	1
2	0	1	1	1	0	0
3	1	1	1	0	0	0

L

E\R	0	1	2	3	4	5
0	1	2	3	1	2	3
1	0	3	2	0	3	1
2	3	0	1	3	0	1
3	2	1	0	2	1	0

A

E\R	0	1	2	3	4	5
0	1	2	3	5	6	7
1	4	3	2	0	7	6
2	3	4	5	7	0	1
3	6	5	4	2	1	0

M

Figure 4.1: Exemple 4 équipes

4.1.2 Contraintes

Cette section présente les contraintes permettant d'obtenir un tournoi à la ronde double qui respecte les contraintes du TTP.

Une équipe ne peut jouer contre elle-même :

$$A_{i,j} \neq i \qquad \forall i, j \tag{4.3}$$

Si une équipe joue contre un adversaire dans une ronde, il faut que cet adversaire joue contre cette même équipe:

$$i = A_{A_{i,j},j} \qquad \forall i, j \tag{4.4}$$

Pour chaque match, une des deux équipes doit être à domicile et l'autre sur la route. Leurs variables de lieu doivent être différentes.

$$L_{i,j} \neq L_{A_{i,j},j} \qquad \forall i,j \tag{4.5}$$

Pour faire un tournoi à la ronde double, il faut que les matchs de chaque équipe soient tous différents, ainsi que les adversaires de chaque ronde:

$$AllDiff(M_{i,j}|0 \leq j \leq 2n-2) \qquad \forall i \tag{4.6}$$

$$AllDiff(A_{i,j}|0 \leq i \leq n-1) \qquad \forall j \tag{4.7}$$

L'interdiction de jouer plus de trois matchs consécutifs à domicile ou à l'extérieur se modélise par une contrainte de type *Stretch* (4.8). Elle stipule que les différentes valeurs du domaine des variables de lieu pour une équipe doivent se trouver en séquences de taille variant de un à trois.

$$Stretch(L_{i,j}|0 \leq j \leq 2n-2, \langle 1,1,...,1\rangle, \langle 3,3,...,3\rangle) \qquad \forall i \tag{4.8}$$

Une équipe ne peut jouer deux fois de suite contre le même adversaire. Cela peut se faire avec la contrainte (4.9). Une contrainte globale de type *Stretch* peut aussi être utilisée, mais cela s'avère moins performant dans ce cas particulier.

$$A_{i.j} \neq A_{i,j+1} \qquad \forall i, j < 2n-3 \tag{4.9}$$

4.2 Modélisation du BSSP

4.2.1 Variables

La modélisation se fait avec les mêmes variables que le TTP; il faut cependant une matrice de constantes booléennes E servant à déterminer si deux équipes sont dans un même état du Brésil. L'intersection d'une ligne i et d'une colonne j indique si les équipes i et j sont géographiquement situées dans le même état du Brésil. Lorsque la valeur est 1, les équipes sont dans le même état et dans le cas contraire la valeur est 0.

4.2.2 Contraintes

Toutes les contraintes du TTP sont réutilisées pour le BSSP. La contrainte (4.9) n'est cependant plus nécessaire, car la structure miroir du BSSP assure de ne jamais jouer deux matchs consécutifs contre le même adversaire. De plus, la contrainte *Stretch* (4.8) pour la longueur des voyages doit être réduite à deux matchs. Les contraintes suivantes doivent être ajoutées pour tenir compte des éléments qui sont différents du TTP.

Soit $H = n/2 - 1$ la moitié du nombre total de rondes, la contrainte (4.10) oblige la deuxième moitié de l'horaire à être identique à la première, alors que la contrainte (4.11) inverse les variables de lieu. Ces deux contraintes font en sorte que la deuxième moitié de l'horaire se définit complètement en fonction de la première. Les prochaines contraintes pourront ainsi être appliquées uniquement à la première

moitié.

$$A_{i,j} = A_{i,j+H} \qquad \forall i, j < H \tag{4.10}$$

$$L_{i,j} \neq L_{i,j+H} \qquad \forall i, j < H \tag{4.11}$$

La contrainte (4.12) assure que les équipes jouent un match à domicile et un match à l'extérieur dans les deux premières rondes de l'horaire, alors que les contraintes (4.13) et (4.14) assurent de donner la configuration inverse de ces premières rondes aux deux dernières de la première moitié.

$$L_{i,0} \neq L_{i,1} \qquad \forall i \tag{4.12}$$

$$L_{i,0} \neq L_{i,H-1} \qquad \forall i \tag{4.13}$$

$$L_{i,1} \neq L_{i,H-2} \qquad \forall i \tag{4.14}$$

La dernière contrainte (4.15) assure que l'adversaire de chaque équipe lors de la dernière ronde de l'horaire (indice $D = 2n - 3$) provient d'un état différent.

$$E_{i,A_{i,D}} = 0 \qquad \forall i \tag{4.15}$$

4.3 Descente à voisinage variable

La structure de notre approche est une descente à voisinage variable. Cette descente utilise cinq voisinages locaux et un grand voisinage. Les cinq voisinages de recherche locale sont :

- *PartialSwapTwoRounds* ($S2R$) : échange de matchs entre deux rondes.
- *PartialSwapTwoTeams* ($S2T$) : échange de matchs entre deux équipes.
- *PartialSwapThreeRounds* ($S3R$) : échange de matchs entre trois rondes.
- *PartialSwapThreeTeams* ($S3T$) : échange de matchs entre trois équipes.
- *NewSwapHomes* (NSH) : échange de lieux.

Les voisinages *PartialSwapTwoRounds* et *PartialSwapTwoTeams* sont deux mouvements provenant de la méthode de recuit simulé proposée par Anagnostopoulos et al. [2]. La mise en oeuvre de ces deux voisinages est très différente de la technique de recuit simulé. Ils sont cependant utilisés tels que proposés. Les voisinages *PartialSwapThreeRounds*, *PartialSwapThreeTeams* sont deux nouveaux voisinages que nous proposons. Le voisinage *NewSwapHomes* est une généralisation du mouvement *SwapHomes* [2]. Ces voisinages sont conçus et explorés selon la technique de recherche locale en programmation par contraintes proposée par Pesant et Gendreau et présentée à la section 2.2.1.

La VND que nous proposons est une recherche qui explore d'abord les voisinages de recherche locale jusqu'à ce que ces derniers aient tous atteint un optimum local. À ce moment, le grand voisinage GV est utilisé pour «dépanner» la recherche, c'est-à-dire l'aider à s'extraire d'un optimum local. Dès qu'une nouvelle meilleure solution est trouvée, le grand voisinage est mis de côté pour retourner aux voisinages plus rapides à explorer.

Plusieurs stratégies d'exploration des voisinages de recherche locale sont possibles. Ces stratégies sont toutes basées sur un schéma de base et une suite ordonnée de voisinages, par exemple : $\langle S2R, S2T, S3R, S3T, NSH \rangle$. Les figures 4.2, 4.3 et 4.4 présentent sous forme d'algorithmes les trois schémas de base que nous utilisons. Dans ces figures, $f(s)$ est une fonction qui retourne la valeur d'une solution s. Nous présentons ensuite les huit stratégies que nous utiliserons.

Définir une suite de voisinages V_i, où $i = 1, 2, ..., i_{max}$
Définir GV le grand voisinage
Trouver une solution initiale s
Faire
  $i \leftarrow 1$
  Tant que $i \leq i_{max}$ **Faire**
    Trouver le meilleur voisin $s' \in V_i(s)$
    Si $f(s') < f(s)$ **Alors**
      $i \leftarrow 1$
      $s \leftarrow s'$
    Sinon
      $i \leftarrow i + 1$
    Fin Si
  Fin Tant que
  Si $GV(s)$ trouve un voisin s' tel que $f(s') < f(s)$ **Alors**
    $s \leftarrow s'$
    continuer $\leftarrow$ VRAI
  Sinon
    continuer$\leftarrow$ FAUX
  Fin Si
Tant que continuer

Figure 4.2: Schéma A

Définir une suite de voisinages V_i, où $i = 1, 2, ..., i_{max}$
Définir GV le grand voisinage
Trouver une solution initiale s
Faire
  $i \leftarrow 1$, mieux $\leftarrow$ FAUX
  Tant que $i \leq i_{max}$ **Faire**
    Trouver le meilleur voisin $s' \in V_i(s)$
    Si $f(s') < f(s)$ **Alors**
      $s \leftarrow s'$, mieux $\leftarrow$ VRAI
    Sinon Si mieux **Alors**
      $i \leftarrow 1$, mieux $\leftarrow$ FAUX
    Sinon
      $i \leftarrow i + 1$
    Fin Si
  Fin Tant que
  Si $GV(s)$ trouve un voisin s' tel que $f(s') < f(s)$ **Alors**
    $s \leftarrow s'$
    continuer $\leftarrow$ VRAI
  Sinon
    continuer$\leftarrow$ FAUX
  Fin Si
Tant que continuer

Figure 4.3: Schéma B

Définir l'ensemble des voisinages V_i, où $i = 1, 2, ..., i_{max}$
Définir GV le grand voisinage
Trouver une solution initiale s
Faire
  $i \leftarrow 1$, *echec* $\leftarrow 0$
  Faire
    Trouver le meilleur voisin $s' \in V_i(s)$
    Si $f(s') < f(s)$ **Alors**
      $s \leftarrow s'$, echec $\leftarrow 0$
    Sinon
      echec $\leftarrow$ echec+1
      Si $i < i_{max}$ **Alors**
        $i \leftarrow i + 1$
      Sinon
        $i \leftarrow 1$
      Fin Si
    Fin Si
  Tant que échec $< i_{max}$
  Si $GV(s)$ trouve un voisin s' tel que $f(s') < f(s)$ **Alors**
    $s \leftarrow s'$
    continuer $\leftarrow$ VRAI
  Sinon
    continuer$\leftarrow$ FAUX
  Fin Si
Tant que continuer

Figure 4.4: Schéma C

- VNDA1 : Schéma A, i_{max}=5, suite $\langle S2R, S2T, S3R, S3T, NSH \rangle$.
  La figure 4.2 montre que la recherche débute toujours par le premier voisinage de la suite. Si l'exploration d'un voisinage V_i mène à une nouvelle meilleure solution, la recherche retourne aussitôt au voisinage V_1 ($S2R$). Sinon, la recherche passe au voisinage suivant dans la liste V_{i+1}, car les voisinages $\langle V_1, V_2, ..., V_i \rangle$ sont tous bloqués dans un optimum local. On remarque que selon cette procédure les voisinages $V_{i>1}$ ne sont jamais explorés plus d'une itération consécutive.

- VNDA2 : Schéma A, i_{max}=5, suite $\langle S2T, S2R, S3T, S3R, NSH \rangle$.
  La procédure est la même que pour VNDA1, mais on utilise une suite différente de voisinages.

- VNDB1 : Schéma B, i_{max}=5, suite $\langle S2R, S2T, S3R, S3T, NSH \rangle$.
  La figure 4.3 montre que la recherche débute toujours par le premier voisinage de la suite. Le schéma B est presque identique au schéma A, la différence est qu'un voisinage V_i est utilisé tant qu'il est capable d'améliorer la solution courante. La recherche peut donc passer plusieurs itérations consécutives avec un voisinage $V_{i>1}$, avant de retourner à V_1.

- VNDB2 : Schéma B, i_{max}=5, suite $\langle S2T, S2R, S3T, S3R, NSH \rangle$.
  La procédure est la même que pour VNDB1, mais on utilise une suite différente de voisinages.

- VNDC1 : Schéma C, i_{max}=5, suite $\langle S2R, S2T, S3R, S3T, NSH \rangle$.
  La figure 4.4 montre que les voisinages sont explorés tant que possible l'un après l'autre. Cette stratégie forme une boucle, car lorsque le dernier voisinage de la suite est atteint, la recherche retourne au premier.

- VNDC2 : Schéma C, i_{max}=5, suite $\langle S2T, S2R, S3T, S3R, NSH \rangle$.

La procédure est la même que pour VNDC1, mais on utilise une suite différente de voisinages.

- VNDD : Schéma A, i_{max}=2, suite $\langle S2R{+}S2T, S3R{+}S3T{+}NSH\rangle$.
  Cette stratégie combine certains voisinages ensemble. Ainsi, les voisinages $S2R$ et $S2T$ sont combinés en un seul voisinage tout comme les voisinages $S3R$, $S3T$ et NSH. Chaque groupe de voisinages est utilisé comme un seul voisinage; chaque voisinage d'un groupe est exploré et le meilleur voisin du groupe est préservé.

- VNDE : Schéma B, i_{max}=2, suite $\langle S2R{+}S2T, S3R{+}S3T{+}NSH\rangle$.
  La procédure est la même que pour VNDD, mais on utilise le schéma B.

4.3.1 Voisinages de recherche locale

Les sections qui suivent présentent le fonctionnement de chacun des mouvements de recherche locale ainsi que leurs modèles en programmation par contraintes. Ces voisinages ont été développés d'abord pour le TTP et réutilisés pour le BSSP. La structure en miroir du BSSP fait cependant en sorte que la seconde moitié du tournoi est complètement dépendante de la première. Nous avons ainsi considéré uniquement la première moitié du tournoi, soit les 23 premières rondes du BSSP.

4.3.1.1 PartialSwapTwoRounds

Le mouvement *PartialSwapTwoRounds* consiste à échanger les matchs de deux rondes différentes pour la même équipe. Les indices t, r_1 et r_2 représentent respectivement l'équipe et les deux rondes. L'idée est donc de permuter les cases $M_{t,r1}$ et $M_{t,r2}$ d'un horaire. Les figures 4.5, 4.6 et 4.7 montrent un exemple de ce mou-

T\R	0	1	2	3	4	**5**	6	7	8	9	10	11	12	**13**
0	@1	@2	@3	1	@4	@5	@6	2	3	4	@7	5	6	7
1	0	@3	@2	@0	2	**@4**	@5	@6	4	7	3	@7	5	**6**
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	@6	@5	6	@2	@7	@0	5	@1	@4	7	4
4	6	7	@5	@2	0	1	@7	5	@1	@0	@6	3	2	@3
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	3	7	@3	0	1	@5	@2	4	2	@0	@1
7	@5	@4	6	5	@6	2	4	3	@2	@1	0	1	@3	@0

Figure 4.5: Solution initiale d'un mouvement $t = 1$, $r_1 = 5$, $r_2 = 13$

T\R	0	1	2	3	4	**5**	6	7	8	9	10	11	12	**13**
0	@1	@2	@3	1	@4	@5	@6	2	3	4	@7	5	6	7
1	0	@3	@2	@0	2	**@4**	@5	@6	4	7	3	@7	5	**6**
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	@6	@5	6	@2	@7	@0	5	@1	@4	7	4
4	6	7	@5	@2	0	**1**	@7	5	@1	@0	@6	3	2	@3
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	3	7	@3	0	1	@5	@2	4	2	@0	**@1**
7	@5	@4	6	5	@6	2	4	3	@2	@1	0	1	@3	@0

Figure 4.6: Adversaires à permuter

vement pour le TTP. Pour simplifier l'analyse, le symbole @ indique des matchs à l'extérieur.

Permuter uniquement les cases $M_{t,r1}$ et $M_{t,r2}$ mène cependant à un horaire non valide. Pour satisfaire les contraintes du tournoi à la ronde (TTP ou BSSP), des cases supplémentaires doivent être permutées. La contrainte (4.4) fait en sorte qu'un match est représenté par deux équipes dans une ronde. À la case $M_{1,5}$ de la figure 4.6, l'adversaire de l'équipe 1 est l'équipe 4; ainsi, le match se définit par les cases $M_{1,5}$ et $M_{4,5}$. Vouloir permuter une des deux équipes d'un match implique donc de permuter aussi l'adversaire. Pour le mouvement $t = 1$, $r_1 = 5$, $r_2 = 13$, il

faut donc permuter les adversaires aux cases $M_{4,5}$ et $M_{6,13}$ avec leurs homologues des rondes opposées, les cases $M_{4,13}$ et $M_{6,5}$. Ces homologues ont eux aussi des adversaires qu'il faut permuter, soient les cases $M_{3,5}$ et $M_{3,15}$. Le mécanisme de recherche des adversaires et homologues à permuter doit se poursuivre jusqu'à l'obtention du sous-ensemble des équipes qu'il faut permuter pour obtenir deux nouvelles rondes valides. Un mouvement *PartialSwapTwoRounds* peut mener à une permutation complète des deux rondes. La figure 4.7 montre que pour le mouvement $t = 1$, $r_1 = 5$, $r_2 = 13$, le sous-ensemble des équipes à permuter est $\{1, 3, 4, 6\}$.

En étudiant ce sous-ensemble, il ressort que choisir n'importe quelle équipe parmi $\{1, 3, 4, 6\}$ mènera toujours vers le même sous-ensemble d'équipes à permuter. Ainsi, chaque sous-ensemble représente une classe de mouvements équivalents. Étant donné deux rondes r_1 et r_2, définissons une relation d'équivalence $\Pi(r_1, r_2)$ départageant l'ensemble des équipes en ces classes de mouvements équivalents. Nous noterons $\Pi(r_1, r_2)[t]$ la classe à laquelle appartient l'équipe t. Ainsi, $\Pi(5, 13)[1] = \{1, 3, 4, 6\}$ dans l'exemple précédent. Sans perte de généralité, désignons le plus petit élément de chaque classe comme son représentant et dénotons par $\rho(r_1, r_2)$ l'ensemble des représentants des classes de la relation d'équivalence $\Pi(r_1, r_2)$.

Voisinage

Le voisinage du mouvement *PartialSwapTwoRounds* est représenté par les trois variables T, R_1 et R_2. Le domaine de T est l'ensemble des équipes et le domaine pour R_1 et R_2 est l'ensemble des rondes du tournoi respectant $R_1 < R_2$. À chaque itération de la recherche locale, il faut trouver parmi ce voisinage la combinaison qui admet la meilleure amélioration à la solution courante. Cette recherche de meilleur voisin prend la forme d'une recherche dans un arbre où chaque noeud de l'arbre correspond normalement à l'affectation d'une des trois variables. Pour le

T\R	0	1	2	3	4	**5**	6	7	8	9	10	11	12	**13**
0	@1	@2	@3	1	@4	@5	@6	2	3	4	@7	5	6	7
1	0	@3	@2	@0	2	**6**	@5	@6	4	7	3	@7	5	**@4**
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	@6	@5	**4**	@2	@7	@0	5	@1	@4	7	**6**
4	6	7	@5	@2	0	**@3**	@7	5	@1	@0	@6	3	2	**1**
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	3	7	**@1**	0	1	@5	@2	4	2	@0	**@3**
7	@5	@4	6	5	@6	2	4	3	@2	@1	0	1	@3	@0

Figure 4.7: État final mouvement $t=1$, $r_1=5$, $r_2=13$

BSSP, nous limitons les variables de rondes à la première moitié de l'horaire.

La taille du voisinage V pour une itération est l'ensemble des combinaisons possibles de $R_1 < R_2$ multiplié par le nombre d'équipes N. Soit κ le nombre de rondes ($2N-2$ pour le TTP et 23 pour le BSSP), la formule (4.16) permet de calculer la taille du voisinage.

$$|V| = N \cdot \frac{\kappa(\kappa-1)}{2} \tag{4.16}$$

Contraintes d'interface

Les contraintes d'interface permettent d'appliquer à une solution courante σ un mouvement de recherche locale menant vers une nouvelle solution. Cette nouvelle solution est définie par ses matchs; ainsi, M sera utilisé pour la représenter.

Lorsqu'une variable ou un sous-ensemble des variables du voisinage est fixé, certaines contraintes d'interface s'activent et instancient une portion de la nouvelle solution. Les contraintes du modèle maître peuvent alors être propagées sur M et dans certains cas, la solution partielle sera rejetée, réduisant ainsi l'espace de recherche du voisinage.

Le mouvement *PartialSwapTwoRounds* apporte des changements uniquement dans les deux rondes choisies; ainsi, le reste de la solution demeure la même. Les contraintes d'interface (4.17), (4.18) et (4.19) servent justement à fixer les trois blocs invariants délimités par les rondes R_1 et R_2. Si la variable R_1 est fixée à la valeur r_1, la contrainte (4.17) stipule alors que chaque variable $M_{i,j}$, où $j < r_1$, doit prendre la valeur de la solution courante $\sigma_{i,j}$. Cette contrainte fixe donc les rondes précédant R_1. La contrainte (4.18) fixe les rondes entre R_1 et R_2 et la contrainte (4.19) fixe les rondes suivant R_2.

$$(R_1 = r_1) \Rightarrow \bigwedge_{\forall (i,j) | j < r_1} M_{i,j} = \sigma_{i,j} \tag{4.17}$$

$$((R_1 = r_1) \wedge (R_2 = r_2)) \Rightarrow \bigwedge_{\forall (i,j) | r_1 < j < r_2} M_{i,j} = \sigma_{i,j} \tag{4.18}$$

$$(R_2 = r_2) \Rightarrow \bigwedge_{\forall (i,j) | j > r_2} M_{i,j} = \sigma_{i,j} \tag{4.19}$$

Nous utilisons E comme symbole pour représenter l'ensemble de toutes les rondes du tournoi pour ne pas porter confusion avec les variables de rondes. La contrainte d'interface (4.20) fait en sorte que pour chaque équipe d'une classe d'équivalence $Pi(r_1, r_2)[t]$, il faut permuter les valeurs des deux rondes, alors que la contrainte d'interface (4.21) garde les mêmes valeurs pour les autres équipes.

$$((T = t) \wedge (R_1 = r_1) \wedge (R_2 = r_2)) \Rightarrow \bigwedge_{\forall i \in \Pi(r_1, r_2)[t]} M_{i,r_1} = \sigma_{i,r_2} \wedge M_{i,r_2} = \sigma_{i,r_1} \tag{4.20}$$

$$((T = t) \wedge (R_1 = r_1) \wedge (R_2 = r_2)) \Rightarrow \bigwedge_{\forall i \in (E \setminus \Pi(r_1, r_2)[t])} M_{i,r_1} = \sigma_{i,r_1} \wedge M_{i,r_2} = \sigma_{i,r_2} \tag{4.21}$$

Typiquement, les contraintes d'interface servent à traduire le mouvement de recherche locale en nouvelle solution. L'arbre de recherche est élagué lorsqu'une affectation partielle du voisinage produit une solution partielle qui est rejetée par les contraintes du modèle maître. Afin de réduire encore plus la taille de l'arbre

de recherche, il est possible d'ajouter des contraintes d'interface qui filtreront le domaine des variables de voisinage non affectées.

L'expression de ces contraintes dépend de l'ordre dans lequel les variables sont affectées. Dans le cas du mouvement *PartialSwapTwoRounds*, lorsque la variable T et qu'une des deux variables de ronde R est fixée, il est possible de réduire le domaine de la variable de ronde non fixée. Si R_1 est la variable fixée, le domaine de M_{T,R_1} est filtré par la contrainte (4.9) pour que $A_{T,R_1} \neq A_{T,R_1-1}$. De plus, dans le cas où les matchs de l'équipe T dans les trois rondes précédant R_1 sont tous à domicile ou à l'extérieur, la contrainte (4.8) filtre du domaine de M_{T,R_1} les matchs pour ne pas en jouer au même lieu. La contrainte d'interface (4.22) permet de récupérer l'effet du filtrage de la variable M_{T,R_1} par le modèle maître et de l'appliquer à la variable R_2, cela en enlevant du domaine de R_2 les valeurs telles que σ_{T,R_2} n'est pas dans le domaine de M_{T,R_2} suite au filtrage.

$$((T=t) \wedge (R_{fixée}=r)) \Rightarrow M_{t,r}=\sigma_{t,R_{non\ fixée}} \tag{4.22}$$

Lorsque ce sont les variables de ronde qui sont affectées d'abord, il est possible de réduire considérablement le voisinage d'une solution. Puisque toutes les équipes d'une classe d'équivalence mènent vers un même mouvement, un seul représentant par classe peut être préservé. La contrainte d'interface (4.23) permet d'éliminer les redondances en ne gardant qu'un seul candidat pour chaque sous-ensemble Π. Pour l'exemple de la figure 4.7, le domaine de T sera réduit à $\rho(r_5, r_{13}) = \{0, 1\}$, éliminant ainsi six des huit valeurs possibles. Sachant qu'un match dans une ronde est représenté par deux équipes jouant l'une contre l'autre et que permuter l'une de ces deux équipes implique de permuter la deuxième, la contrainte (4.23) réduit au moins de moitié le domaine de T pour tout échange et pour toute taille de problème.

$$((R_1=r_1) \wedge (R_2=r_2)) \Rightarrow T \in \rho(r_1, r_2) \tag{4.23}$$

Nouvelle forme des contraintes d'interface

La technique proposée par Pesant et Gendreau utilise des conditions de réveil dans l'expression des contraintes. Ces conditions sont toutes basées sur les événements *WhenValue*; lorsque les variables impliquées sont affectées à une valeur la contrainte peut être réveillée. Nous avons exprimé toutes les contraintes du voisinage *PartialSwapTwoRounds* de la section précédente de cette façon.

Nous proposons une nouvelle forme de contraintes d'interface sans conditions de réveils. L'idée est qu'il peut être utile d'activer ces contraintes après une simple modification du domaine d'une des variables impliquées. Cette modification peut être l'élimination d'une seule valeur du domaine, un resserrement des bornes ou l'affectation à une valeur. Cette nouvelle forme de contrainte a pour but d'utiliser plus activement les contraintes en incitant la recherche d'algorithme de filtrage pour des modifications plus élémentaires des domaines.

Nous utiliserons désormais cette nouvelle forme pour les contraintes d'interface des prochains voisinages. Nous allons avant cela reprendre les contraintes du voisinage *PartialSwapTwoRounds* et les exprimer sous la nouvelle forme et ensuite discuter quels traitements nous pouvons offrir selon les modifications du domaine des variables.

$$\bigwedge_{\forall(i,j)|j<R_1} M_{i,j} = \sigma_{i,j} \tag{4.24}$$

$$\bigwedge_{\forall(i,j)|R_1<j<R_2} M_{i,j} = \sigma_{i,j} \tag{4.25}$$

$$\bigwedge_{\forall(i,j)|j>R_2} M_{i,j} = \sigma_{i,j} \tag{4.26}$$

En étudiant les contraintes (4.24), (4.25) et (4.26) on réalise que l'on peut faire des opérations lorsque les bornes du domaine de R_1 ou de R_2 changent. En effet, on peut fixer dans l'horaire les rondes se trouvant avant la borne inférieure de R_1,

celles qui sont entre la borne supérieure de R_1 et la borne inférieure de R_2 et celles qui sont après la borne supérieure de R_2. On peut alors réveiller ces contraintes sur les événement $WhenRange$. Nous avons implémenté ces contraintes pour les événements $WhenValue$ et $WhenRange$, nous les comparons les résultats obtenus dans le cinquième chapitre.

$$\bigwedge_{\forall i \in \Pi_{R_1,R_2}[T]} M_{i,R_1} = \sigma_{i,R_2} \wedge M_{i,R_2} = \sigma_{i,R_1} \tag{4.27}$$

$$\bigwedge_{\forall i \in (E \setminus \Pi_{R_1,R_2}[T])} M_{i,R_1} = \sigma_{i,R_1} \wedge M_{i,R_2} = \sigma_{i,R_2} \tag{4.28}$$

Notre implémentation attend que les variables soient toutes affectées pour réveiller les contraintes (4.27) et (4.28). Nous avons cependant pensé à un traitement lorsque les bornes du domaine de R_1 ou de R_2 changent. Par exemple, on coupe le domaine de R_1 en deux. Le nouveau domaine est R_1=[0,1,2,3] et les valeurs éliminées sont [4,5,6]. Le domaine de R_2 est [1,2,3,4,5,6,7]. Il n'y a que R_2 qui peut alors prendre pour valeur les rondes [4,5,6,7]. Les valeurs possibles pour un match dans une ronde éliminée sont : la valeur de ce match dans solution courante pour la cas où on ne le modifierait pas et la valeur des matchs des rondes dans le domaine de R_1 pour le cas où on le modifierait. Ainsi on peut enlever les valeurs :

$\sigma_{1,4}$ aux domaines des matchs $M_{1,5}$, $M_{1,6}$, $M_{1,7}$,
$\sigma_{1,5}$ aux domaines des matchs $M_{1,4}$, $M_{1,6}$, $M_{1,7}$,
$\sigma_{1,6}$ aux domaines des matchs $M_{1,4}$, $M_{1,5}$, $M_{1,7}$,
$\sigma_{1,7}$ aux domaines des matchs $M_{1,4}$, $M_{1,5}$, $M_{1,6}$.

On remarque que, dans une ronde, il ne faut pas enlever la valeur de la solution courante du domaine de la variable de match. Ce traitement permet ainsi de réduire les domaines des variables de matchs. Il y a évidemment un traitement équivalent lorsque ce sont les bornes du domaine de la variable R_2 qui changent.

Pour la contrainte (4.22), nous avions déjà discuté de la possibilité de filtrer la variable de match qui n'est pas encore affectée. La nouvelle forme de cette contrainte devient un cas particulier de la contrainte (4.27) où $i = T$. Cette contrainte a pour but d'assurer que l'échange soit valide. On peut donc aussi réveiller la contrainte lorsque les variables de rondes sont affectées et filtrer T pour choisir les valeurs qui mènent à un mouvement valide. Nous n'avons cependant pas implémenté ce cas particulier de la contrainte.

$$T \in \rho(R_1, R_2) \tag{4.29}$$

Notre implémentation attend que les variables R_1 et R_2 soient affectées pour réveiller la contrainte (4.29) qui filtre le domaine de T.

4.3.1.2 PartialSwapTwoTeams

Le mouvement *PartialSwapTwoTeams* consiste à permuter au sein d'une même ronde les matchs de deux équipes différentes. Les indices T_a, T_b et R représentent respectivement les deux équipes et la ronde. Les matchs $M_{T_a,R}$ et $M_{T_b,R}$ seront permutés. Un peu comme le mouvement *PartialSwapRounds*, il faut faire des réparations. Pour l'exemple de la figure 4.8, la figure 4.9 présente toutes les autres rondes où il faut permuter les matchs des équipes T_a, T_b pour que ces deux équipes jouent contre toutes les autres équipes à deux reprises. Dans toutes ces rondes, il faut aussi mettre à jour les matchs des adversaires des équipes T_a et T_b de façon à créer de nouvelles rondes valides. Dans la ronde 5, l'équipe 1 jouait au domicile de l'équipe 4. La figure 4.10 montre que le nouveau match de l'équipe 1 est contre l'équipe 6, mais au domicile de l'équipe 1. La permutation implique donc qu'il faut aussi modifier le match de l'équipe 6 qui devient alors $M_{6,5} = 1$.

Ici encore, l'ensemble des rondes affectées par un mouvement *PartialSwapTwoTeams* est une classe d'équivalence; choisir n'importe quelle valeur pour R dans

T\R	0	1	2	3	4	**5**	6	7	8	9	10	11	12	13
0	@1	@2	@3	1	@4	@5	@6	2	3	4	@7	5	6	7
1	0	@3	@2	@0	2	**@4**	@5	@6	4	7	3	@7	5	6
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	@6	@5	**6**	@2	@7	@0	5	@1	@4	7	4
4	6	7	@5	@2	0	1	@7	5	@1	@0	@6	3	2	@3
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	3	7	@3	0	1	@5	@2	4	2	@0	@1
7	@5	@4	6	5	@6	2	4	3	@2	@1	0	1	@3	@0

Figure 4.8: Mouvement $T_a = 5$ T_b=1 R=3

T\R	0	1	2	3	4	**5**	6	7	8	9	10	11	12	13
0	@1	@2	@3	1	@4	@5	@6	2	3	4	@7	5	6	7
1	0	@3	@2	**@0**	2	**@4**	@5	**@6**	**4**	7	3	**@7**	5	**6**
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	**@6**	@5	**6**	@2	**@7**	**@0**	5	@1	**@4**	7	**4**
4	6	7	@5	@2	0	1	@7	5	@1	@0	@6	3	2	@3
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	3	7	@3	0	1	@5	@2	4	2	@0	@1
7	@5	@4	6	5	@6	2	4	3	@2	@1	0	1	@3	@0

Figure 4.9: Rondes affectées par le mouvement

une classe d'équivalence mène toujours à permuter toutes les rondes de cette classe. Il sera donc possible de réduire le domaine de R avec une contrainte d'interface. De plus, il sera impossible de choisir R où l'adversaire de l'équipe T_a est l'équipe T_b, car l'échange mènerait à des matchs où les équipes T_a et T_b joueraient contre elles-mêmes. Il s'agit des rondes 1 et 10 pour le mouvement de la figure 4.8.

Voisinage

Le voisinage du mouvement *PartialSwapTwoTeams* est représenté par les trois variables T_a, T_b et R. Le domaine de T_a et T_b est l'ensemble des équipes respectant

T\R	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	@1	@2	@3	**3**	@4	@5	@6	2	**1**	4	@7	5	6	7
1	0	@3	@2	**@6**	2	**6**	@5	**@7**	**@0**	7	3	**@4**	5	**4**
2	@3	0	1	4	@1	@7	3	@0	7	6	@5	@6	@4	5
3	2	1	0	**@0**	@5	**@4**	@2	**@6**	**4**	5	@1	**@7**	7	**6**
4	6	7	@5	@2	0	**3**	@7	5	**@3**	@0	@6	**1**	2	**@1**
5	7	@6	4	@7	3	0	1	@4	6	@3	2	@0	@1	@2
6	@4	5	@7	**1**	7	**@1**	0	**3**	@5	@2	4	2	@0	**@3**
7	@5	@4	6	5	@6	2	4	**1**	@2	@1	0	**3**	@3	@0

Figure 4.10: Mise à jour des adversaires

$T_a < T_b$ et le domaine de R est l'ensemble des rondes du tournoi. Pour le BSSP, nous limitons la variable de rondes à la première moitié de l'horaire.

La taille du voisinage V pour une itération est l'ensemble des combinaisons possibles de $T_a < T_b$ multiplié par le nombre de rondes R. Soit κ le nombre de rondes ($2n-2$ pour le TTP et 23 pour le BSSP), la formule (4.30) permet de calculer la taille du voisinage.

$$|V| = \frac{(n)(n-1)}{2} \cdot \kappa \tag{4.30}$$

Contraintes d'interface

Nous exprimons les contraintes de ce voisinage selon la nouvelle forme. Les traitements que nous offrons pour toutes ces contraintes sont cependant activés lorsque toutes les variables de voisinage impliquées sont affectées.

Comme il a été mentionné précédemment, les classes d'équivalence permettent de réduire considérablement le domaine de R lorsque les variables T_a et T_b sont déterminées. En reprenant la notation établie à la section précédente, soit $\rho(T_a, T_b)$ l'ensemble des représentants de chacune des classes d'équivalence admises pour un échange de matchs des équipes T_a et T_b. On ne retient pas les rondes où l'équipe

T_a est l'adversaire de l'équipe T_b, car ces matchs ne peuvent être permutés. La contrainte d'interface (4.31) permet d'éliminer les redondances en ne gardant qu'un seul candidat pour chaque classe d'équivalence valide. Pour l'exemple de la figure 4.8, les classes d'équivalence de R sont $\langle\{0,2,4,6\},\{3,5,7,8,11,13\},\{9,12\}\rangle$ et les matchs des rondes 1 et 10 ne peuvent être permutées. L'ensemble des représentants est $\rho(1,3)=\{0,3,9\}$, le domaine réduit de R.

$$R \in \rho(T_a, T_b) \tag{4.31}$$

Soit maintenant E l'ensemble de toutes les équipes, $\Pi(T_a, T_b)[R]$ l'ensemble des rondes d'une classe d'équivalence pour un mouvement $PartialSwapTwoTeams$ et σ la solution courante. La contrainte d'interface (4.32) fait en sorte que pour toutes les rondes où il n'y a pas de permutation, la nouvelle solution prend la valeur de la solution courante. La contrainte d'interface (4.33) applique la permutation des matchs des équipes T_a et T_b dans les rondes de la classe d'équivalence $\Pi(T_a, T_b)[R]$.

$$\bigwedge_{\forall i \forall j \in (E \setminus \Pi(T_a,T_b)[R])} M_{i,j} = \sigma_{i,j} \tag{4.32}$$

$$\bigwedge_{\forall j \in \Pi(T_a,T_b)[R]} (M_{T_a,j} = \sigma_{T_b,j}) \wedge (M_{T_b,j} = \sigma_{T_a,j}) \tag{4.33}$$

$$\bigwedge_{\forall j \in \Pi(T_a,T_b)[R]} (M_{ad[\sigma_{Ta,j}],j} = T_b + (1 - l[\sigma_{Ta,j}]) \cdot n) \wedge (M_{ad[\sigma_{T_b,j}],j} = T_a + (1 - l[\sigma_{T_b,j}]) \cdot n) \tag{4.34}$$

$$\bigwedge_{\forall i | i \notin \{T_a, T_b, ad[\sigma_{T_a,j}], ad[\sigma_{T_b,j}]\} \wedge \forall j | j \in \Pi(T_a,T_b)[R]} M_{i,j} = \sigma_{i,j} \tag{4.35}$$

La contrainte d'interface (4.34) sert à mettre à jour les adversaires des équipes T_a et T_b suite aux permutations. Prenons la ronde 5 de l'exemple, le match de la solution courante $\sigma_{1,5} = @4$ est déplacé contre l'équipe 3. C'est donc dire que l'équipe 4 joue toujours à domicile, mais maintenant contre l'équipe 3. Il faut

donc mettre à jour le match $\sigma_{4,5}$ pour refléter ce changement. On remarque que l'adversaire de l'équipe 4 jouera toujours sur la route dans la ronde 5 car le match @4 est permuté sans changement de localisation. Cela signifie que le lieu du match $\sigma_{4,5}$ sera préservé pour le nouveau match $M_{4,5}$. Il en est de même pour le match $\sigma_{6,5}$. En se souvenant de la convention du @, on sait que pour obtenir la valeur réelle des matchs qui ne sont pas précédés du symbole @ il faut leur ajouter la valeur n. Ainsi par exemple $M = @4 \rightarrow M = 4$ et $M = 4 \rightarrow M = 4 + 8 = 12$. De plus, en sachant que les lieux des matchs des adversaires sont préservés, on peut utiliser ces lieux pour définir les nouveaux matchs des adversaires. Ainsi, le nouvel adversaire de l'équipe 4 est l'équipe 3 et le match reste au domicile de l'équipe 4 ($l[\sigma_{4,5}] = 1$), on doit donc ajouter n à 3 pour obtenir la nouvelle valeur réelle de $M_{4,5} = 3 + n = 11$. La contrainte (4.34) utilise cette astuce pour définir les nouveaux matchs des adversaires. Pour déterminer les lieux matchs des adversaires dans la solution courante, elle prend l'inverse des lieux des équiques T_a et T_b. Ainsi $M_{4,5} = 3 + (1 - l[\sigma_{1,5}]) \cdot n = 11$.

Finalement, la contrainte d'interface (4.35) affecte les valeurs de la solution courante aux autres matchs des rondes de la classe d'équivalence $\Pi(T_a, T_b)[R]$, soit, pour chacune de ces rondes, les équipes autres que T_a, T_b et leurs adversaires.

4.3.1.3 PartialSwapThreeRounds

Le mouvement *PartialSwapThreeRounds* consiste à appliquer des échanges entre trois rondes R_1, R_2 et R_3 de la solution courante. Cette opération se fait en deux temps. D'abord, pour une équipe T_a, faire un échange de type *PartialSwapTwo-Rounds* sur les rondes R_1 et R_2 pour obtenir deux nouvelles rondes R'_1 et R'_2. Ensuite, pour une équipe T_b, faire un deuxième mouvement *PartialSwapTwoRounds* en utilisant R'_2 et R_3 pour obtenir R''_2 et R'_3. La nouvelle solution sera composée des

T\R	0	1	**2**	3	**4**	5	6	**7**	8	9	10	11	12	13
0	@5	@6	3	4	@1	@2	@3	1	5	6	@7	2	7	@4
1	@4	7	2	@3	0	4	@2	@0	3	@7	@6	@5	6	5
2	7	5	@1	@7	3	0	1	@3	@6	@5	4	@0	@4	6
3	@6	@4	@0	1	@2	6	0	2	@1	4	@5	@7	5	7
4	1	3	@5	@0	5	@1	@7	@6	7	@3	@2	6	2	0
5	0	@2	4	6	@4	7	@6	@7	@0	2	3	1	@3	@1
6	3	0	**@7**	@5	**7**	@3	5	4	2	@0	1	@4	@1	@2
7	@2	@1	**6**	2	**@6**	@5	4	5	@4	1	0	3	@0	@3

Figure 4.11: Première phase d'un mouvement *PartialSwapThreeRounds*

T\R	0	1	**2**	3	**4**	5	6	**7**	8	9	10	11	12	13
0	@5	@6	3	4	**@1**	@2	@3	**1**	5	6	@7	2	7	@4
1	@4	7	2	@3	**0**	4	@2	**@0**	3	@7	@6	@5	6	5
2	7	5	@1	@7	3	0	1	@3	@6	@5	4	@0	@4	6
3	@6	@4	@0	1	@2	6	0	2	@1	4	@5	@7	5	7
4	1	3	@5	@0	5	@1	@7	@6	7	@3	@2	6	2	0
5	0	@2	4	6	@4	7	@6	@7	@0	2	3	1	@3	@1
6	3	0	7	@5	@7	@3	5	4	2	@0	1	@4	@1	@2
7	@2	@1	@6	2	6	@5	4	5	@4	1	0	3	@0	@3

Figure 4.12: Deuxième phase d'un mouvement *PartialSwapThreeRounds*

rondes R'_1, R''_2 et R'_3. L'idée est de permettre une détérioration de la solution lors de la première phase du mouvement, mais de chercher une amélioration à la fin de la deuxième. Les figures 4.11, 4.12 et 4.13 montrent un exemple de ce mouvement.

Voisinage

Le voisinage du mouvement *PartialSwapThreeRounds* est représenté par cinq variables T_a, T_b, R_1, R_2 et R_3 respectivement pour les équipes et les rondes. Il sera possible de réduire le domaine des variables d'équipes grâce aux classes

T\R	0	1	**2**	3	**4**	5	6	**7**	8	9	10	11	12	13
0	@5	@6	3	4	1	@2	@3	@1	5	6	@7	2	7	@4
1	@4	7	2	@3	@0	4	@2	0	3	@7	@6	@5	6	5
2	7	5	@1	@7	3	0	1	@3	@6	@5	4	@0	@4	6
3	@6	@4	@0	1	@2	6	0	2	@1	4	@5	@7	5	7
4	1	3	@5	@0	5	@1	@7	@6	7	@3	@2	6	2	0
5	0	@2	4	6	@4	7	@6	@7	@0	2	3	1	@3	@1
6	3	0	7	@5	@7	@3	5	4	2	@0	1	@4	@1	@2
7	@2	@1	@6	2	6	@5	4	5	@4	1	0	3	@0	@3

Figure 4.13: État final d'un mouvement *PartialSwapThreeRounds*

d'équivalence et d'éliminer des symétries en posant $R_1 < R_2 < R_3$. Pour le BSSP, nous limitons les variables de rondes à la première moitié de l'horaire.

La taille du voisinage V pour une itération est l'ensemble des combinaisons possibles de $T_a < T_b$ multiplié par le nombre de combinaisons possibles de $R_1 < R_2 < R_3$. Soit κ le nombre de rondes ($2n - 2$ pour le TTP et 23 pour le BSSP), la formule (4.36) permet de calculer la taille du voisinage.

$$|V| = \frac{n(n-1)}{2} \cdot \frac{\kappa(\kappa-1)(\kappa-2)}{6} \tag{4.36}$$

Contraintes d'interface

Le mouvement *PartialSwapThreeRounds* apporte des changements uniquement dans les trois rondes choisies, ainsi le reste de la solution demeure le même. Les contraintes d'interface (4.37), (4.38), (4.39) et (4.40) servent à fixer les quatre blocs invariants délimités par les rondes R_1, R_2 et R_2. Lorsque la variable R_1 est fixée à la valeur r_1, la contrainte d'interface (4.37) stipule alors que chaque variable $M_{i,j}$ où $j < r_1$ doit prendre la valeur de la solution courante $\sigma_{i,j}$. Cette contrainte fixe donc les rondes précédant R_1. La contrainte d'interface (4.38) fixe les rondes entre

R_1 et R_2, la contrainte d'interface (4.39) les rondes entre R_2 et R_3 et la contrainte d'interface (4.40) les rondes suivant R_3.

$$\bigwedge_{\forall(i,j)|j<R_1} M_{i,j} = \sigma_{i,j} \tag{4.37}$$

$$\bigwedge_{\forall(i,j)|R_1<j<R_2} M_{i,j} = \sigma_{i,j} \tag{4.38}$$

$$\bigwedge_{\forall(i,j)|R_2<j<R_3} M_{i,j} = \sigma_{i,j} \tag{4.39}$$

$$\bigwedge_{\forall(i,j)|j>R_3} M_{i,j} = \sigma_{i,j} \tag{4.40}$$

On peut activer ces contraintes pour une modification aux bornes des domaines des variables de voisinage impliquées. Le traitement est une adaptation pour un échange à trois rondes de celui proposé pour le voisinage *PartialSwapTwoRounds*.

Les prochaines contraintes de ce voisinage sont toutes activées lorsque les variables impliquées sont toutes affectées. La contrainte d'interface (4.41) fait en sorte que pour chaque équipe de la classe d'équivalence $\Pi(R_1, R_2)[T_a])$, il faut permuter les valeurs des deux rondes alors que la contrainte d'interface (4.42) garde les mêmes valeurs pour les autres équipes. La ronde R_2 n'est pas mise à jour dans la nouvelle solution M, mais plutôt dans une ronde temporaire R'. Les contraintes d'interface (4.43) et (4.44) appliquent le second échange pour la classe d'équivalence de l'équipe T_b entre la ronde R' et la ronde R_3. C'est à ce second échange que la ronde R_2 de la nouvelle solution est fixée.

$$\bigwedge_{\forall i\in\Pi(R_1,R_2)[T_a]} M_{i,R_1} = \sigma_{i,R_2} \wedge R'_i = \sigma_{i,R_1} \tag{4.41}$$

$$\bigwedge_{\forall i\in(E\setminus\Pi(R_1,R_2)[T_a])} M_{i,R_1} = \sigma_{i,R_1} \wedge R'_i = \sigma_{i,R_2} \tag{4.42}$$

$$\bigwedge_{\forall i \in \Pi(R', R_3)[T_b]} M_{i,R_2} = \sigma_{i,R_3} \wedge M_{i,R_3} = R'_i \tag{4.43}$$

$$\bigwedge_{\forall i \in E \setminus \Pi(R', R_3)[T_b]} M_{i,R_2} = R'_i \wedge M_{i,R_3} = \sigma_{i,R_3} \tag{4.44}$$

Les classes d'équivalence permettent de réduire les domaines des variables T_a et T_b; la contrainte d'interface (4.45) pour T_a et la contrainte d'interface (4.46) pour T_b. Nous activons ces contraintes lorsque les variables de rondes impliquées sont affectées et que R' est défini.

$$T_a \in \theta(R_1, R_2) \tag{4.45}$$

$$T_b \in \theta(R', R_3) \tag{4.46}$$

4.3.1.4 PartialSwapThreeTeams

Le mouvement *PartialSwapThreeTeams* consiste à appliquer des échanges entre trois équipes T_a, T_b et T_c de la solution courante. Cette opération se fait en deux temps. D'abord, pour une ronde R_1 et les équipes T_a et T_b, faire un échange de type *PartialSwapTwoTeams* en gardant le résultat dans un tournoi temporaire.

Ensuite, pour une ronde R_2 et les équipes T_b et T_c, faire un deuxième mouvement *PartialSwapTwoTeams* en utilisant le tournoi résultant du premier mouvement. L'idée est de permettre une détérioration de la solution lors de la première phase du mouvement, mais de chercher une amélioration à la fin de la deuxième.

Les figures 4.14, 4.15 et 4.16 montrent un exemple de ce mouvement.

Voisinage

Le voisinage du mouvement *PartialSwapThreeTeams* est représenté par cinq vari-

T\R	0	**1**	2	3	4	**5**	6	7	8	9	10	11	12	13
0	@1	2	3	4	@6	**@4**	**@3**	1	5	6	@7	@5	7	@2
1	0	@4	2	@3	7	@2	@5	@0	3	@7	@6	4	6	5
2	3	@0	@1	@7	5	1	6	@3	@6	4	@5	7	@4	0
3	@2	@6	@0	1	4	**@7**	**0**	2	@1	5	@4	6	@5	7
4	5	1	@5	@0	@3	**0**	**@7**	6	7	@2	3	@1	2	@6
5	@4	7	4	6	@2	@6	1	@7	@0	@3	2	0	3	@1
6	7	3	@7	@5	0	5	@2	@4	2	@0	1	@3	@1	4
7	@6	@5	6	2	@1	**3**	**4**	5	@4	1	0	@2	@0	@3

Figure 4.14: Première phase d'un mouvement *PartialSwapThreeTeams*

T\R	0	**1**	2	3	4	**5**	6	7	8	9	10	11	12	13
0	@1	2	3	**4**	@6	@3	**@4**	1	**5**	6	@7	**@5**	7	@2
1	0	**@4**	2	@3	7	@2	**@5**	@0	3	@7	@6	**4**	6	**5**
2	3	@0	@1	@7	5	1	6	@3	@6	4	@5	7	@4	0
3	@2	@6	@0	1	4	0	@7	2	@1	5	@4	6	@5	7
4	5	**1**	@5	**@0**	@3	@7	**0**	**6**	**7**	@2	3	**@1**	2	**@6**
5	@4	**7**	4	**6**	@2	@6	**1**	**@7**	**@0**	@3	2	**0**	3	**@1**
6	7	3	@7	**@5**	0	5	@2	**@4**	2	@0	1	@3	@1	**4**
7	@6	**@5**	6	2	@1	4	3	**5**	**@4**	1	0	@2	@0	@3

Figure 4.15: Deuxième phase d'un mouvement *PartialSwapThreeTeams*

T\R	0	**1**	2	3	4	**5**	6	7	8	9	10	11	12	13
0	@1	2	3	5	@6	@3	@5	1	4	6	@7	@4	7	@2
1	0	@5	2	@3	7	@2	@4	@0	3	@7	@6	5	6	4
2	3	@0	@1	@7	5	1	6	@3	@6	4	@5	7	@4	0
3	@2	@6	@0	1	4	0	@7	2	@1	5	@4	6	@5	7
4	5	7	@5	6	@3	@6	1	@7	@0	@2	3	0	2	@1
5	@4	1	4	@0	@2	@7	0	6	7	@3	2	@1	3	@6
6	7	3	@7	@4	0	4	@2	@5	2	@0	1	@3	@1	5
7	@6	@4	6	2	@1	5	3	4	@5	1	0	@2	@0	@3

Figure 4.16: État final d'un mouvement *PartialSwapThreeTeams*

ables T_a, T_b, T_c, R_1 et R_2 respectivement pour les équipes et les rondes. Il sera possible de réduire le domaine des variables d'équipes grâce aux classes d'équivalence et d'éliminer des symétries en posant $T_a < T_b < T_c$. Pour le BSSP, nous limitons les variables de rondes à la première moitié de l'horaire.

La taille du voisinage V pour une itération est l'ensemble des combinaisons possibles de $T_a < T_b < T_c$ multiplié par le nombre de combinaisons possibles de $R_1 < R_2$. Soit κ le nombre de rondes ($2n - 2$ pour le TTP et 23 pour le BSSP), la formule (4.47) permet de calculer la taille du voisinage.

$$|V| = \frac{n(n-1)(n-2)}{6} \cdot \frac{\kappa(\kappa-1)}{2} \tag{4.47}$$

Contraintes d'interface

Nous exprimons les contraintes de ce voisinage selon la nouvelle forme. Les traitements que nous offrons pour toutes ces contraintes sont cependant activés lorsque toutes les variables de voisinage impliquées sont affectées.

Comme ce mouvement est une succession de deux mouvements *PartialSwapTwoTeams*, les contraintes d'interface sont donc essentiellement les mêmes. La seule différence est que le premier mouvement crée un tournoi à la ronde temporaire σ' et que le second mouvement réutilise ce tournoi comme solution courante. Lorsque les variables T_a, T_b et R_1 sont fixées, les contraintes d'interface (4.48), (4.49), (4.50), (4.51) sont activées et définissent l'horaire temporaire σ'. Lorsque les variables T_c et R_2 sont fixées et que σ' est définie, les contraintes (4.52), (4.53), (4.54), (4.55)

affectent les valeurs de l'horaire final M.

$$\bigwedge_{\forall i \forall j \in (E \setminus \Pi(T_a,T_b)[R_1])} \sigma'_{i,j} = \sigma_{i,j} \tag{4.48}$$

$$\bigwedge_{\forall j \in \Pi(T_a,T_b)[R_1]} (\sigma'_{T_a,j} = \sigma_{T_b,j}) \wedge (\sigma'_{T_b,j} = \sigma_{T_a,j}) \tag{4.49}$$

$$\bigwedge_{\forall j \in \Pi(T_a,T_b)[R_1]} (\sigma'_{ad[\sigma_{T_a,j}],j} = T_b + (1 - l[\sigma_{T_a,j}]) \cdot n) \wedge$$
$$(\sigma'_{ad[\sigma_{T_b,j}],j} = T_a + (1 - l[\sigma_{Tb,j}]) \cdot n) \tag{4.50}$$

$$\bigwedge_{\forall i | i \notin \{T_a,T_b,ad[\sigma_{T_a,j}],ad[\sigma_{T_b,j}]\} \wedge \forall j | j \in \Pi(T_a,T_b)[R_1]} \sigma'_{i,j} = \sigma_{i,j} \tag{4.51}$$

$$\bigwedge_{\forall i \forall j \in (E \setminus \Pi(T_b,T_c)[R_2])} M_{i,j} = \sigma'_{i,j} \tag{4.52}$$

$$\bigwedge_{\forall j \in \Pi(T_b,T_c)[R_2]} (M_{T_b,j} = \sigma'_{T_c,j}) \wedge (M_{T_c,j} = \sigma'_{T_b,j}) \tag{4.53}$$

$$\bigwedge_{\forall j \in \Pi(T_b,T_c)[R_2]} (M_{ad[\sigma'_{T_a,j}],j} = T_c + (1 - l[\sigma'_{T_a,j}]) \cdot n) \wedge$$
$$(M_{ad[\sigma'_{tb,j}],j} = t_b + (1 - l[\sigma'_{tb,j}]) \cdot n) \tag{4.54}$$

$$\bigwedge_{\forall i | i \notin \{t_b,t_c,ad[\sigma'_{t_b,j}],ad[\sigma'_{t_c,j}]\} \wedge \forall j | j \in \Pi(T_b,T_c)[R_2]} M_{i,j} = \sigma'_{i,j} \tag{4.55}$$

Pour profiter des classes d'équivalence, la contrainte 4.56 filtre le domaine de R_1 lors que T_a et T_b sont affectées. La contrainte 4.57 filtre le domaine de R_2 lors que T_b et T_c sont affectées et que σ' est définie.

$$R_1 \in \rho(T_a, T_b) \tag{4.56}$$

$$R_2 \in \rho(\sigma'[T_b], T_c) \tag{4.57}$$

4.3.1.5 NewSwapHomes

Le voisinage *SwapHomes* proposé par Anagnostopoulos et al. [2] consiste à choisir, dans l'horaire d'une équipe T_a, les deux matchs contre une équipe T_b et à inverser les lieux de ces matchs. Par exemple, inverser les lieux des deux matchs contre l'équipe 5 dans l'horaire de l'équipe 0 de la figure 4.17. Il faudra faire la même modification pour les matchs contre l'équipe 0 dans l'horaire de l'équipe 5.

Le voisinage *NewSwapHomes* que nous proposons est une généralisation du voisinage *SwapHomes*. Le principe est de permettre l'échange entre les lieux de n'importe quel couple de matchs d'une équipe et non plus juste les couples de matchs contre le même adversaire. Ce nouveau voisinage est beaucoup plus grand et inclut le voisinage *SwapHomes*. On choisit uniquement les couples où un match est sur la route et l'autre à domicile, sinon il n'y aura pas de modification à l'horaire. Les figures 4.17, 4.18 et 4.19 montrent un exemple complet de ce mouvement.

L'exemple sur les figures consiste à échanger le lieu du match $\sigma_{0,0} = @1$ avec celui du match $\sigma_{0,3} = 5$. Ainsi les nouveaux matchs sont $M_{0,0} = 1$ et $M_{0,3} = @5$, mais l'équipe 0 a déjà ces matchs planifiés aux rondes 6 et 7. Il faut donc aussi échanger les lieux de ces matchs. Choisir les rondes 0 et 3 est donc équivalent à choisir les rondes 6 et 7. Il s'agit donc d'une classe d'équivalence, mais qui est plus difficle à exploiter et nous n'avons malheureusement pas eu le temps de trouver une méthode pour en tirer profit. On réalise que le mouvement *NewSwapHomes* équivaut à

faire deux mouvements *SwapHomes* en même temps : échanger les lieux des deux matchs contre l'équipe 1 et ceux contre l'équipe 5 dans l'horaire de l'équipe 0. Le mouvement *NewSwapHomes* n'équivaut cependant pas à faire deux mouvements *SwapHomes* consécutifs. Par exemple, on ne peut permuter les lieux des matchs contre l'équipe 1 en premier, car une séquence de quatre matchs sur la route serait alors obtenue. Il faudrait donc permuter les matchs contre l'équipe 5 d'abord dans le mouvement *SwapHomes* et ensuite faire un second mouvement *SwapHomes* pour les matchs contre l'équipe 1. Cependant, le premier mouvement sera rejeté s'il détériore la solution, alors que le second pourrait mener à une solution meilleure que la solution courante.

Comme nous l'avons mentionné au début de cette section sur les voisinages de recherche locale, notre façon de procéder pour le BSSP est de considérer uniquement la première moitié de son horaire lors de la recherche locale. Cette approche permet de réutiliser les voisinages tels que conçus pour le TTP. Cependant, la structure du BSSP est telle que dans l'horaire d'une équipe T_a, les deux matchs contre une équipe T_b se retrouvent aux mêmes rondes des deux moitiés du tournoi. Utiliser le mouvement *NewSwapHomes* sur la première moitié de l'horaire n'est alors pas possible. Nous proposons pour le BSSP d'utiliser le voisinages *SwapHomes* qui consistera à simplement changer le lieu d'un match dans la première moitié de l'horaire. Le match homologue dans la seconde moitié de l'horaire sera modifé par les contraintes du modèle.

Voisinage

Le voisinage du mouvement *NewSwapHomes* est représenté par les trois variables T, R_1 et R_2. Le domaine de T est l'ensemble des équipes et le domaine pour R_1 et R_2 est l'ensemble des rondes du tournoi respectant $R_1 < R_2$.

T\R	**0**	1	2	**3**	4	5	6	7	8	9	10	11	12	13
0	**@1**	2	3	**5**	@6	@3	@5	1	4	6	@7	@4	7	@2
1	0	@5	2	@3	7	@2	@4	@0	3	@7	@6	5	6	4
2	3	@0	@1	@7	5	1	6	@3	@6	4	@5	7	@4	0
3	@2	@6	@0	1	4	0	@7	2	@1	5	@4	6	@5	7
4	5	7	@5	6	@3	@6	1	@7	@0	@2	3	0	2	@1
5	@4	1	4	@0	@2	@7	0	6	7	@3	2	@1	3	@6
6	7	3	@7	@4	0	4	@2	@5	2	@0	1	@3	@1	5
7	@6	@4	6	2	@1	5	3	4	@5	1	0	@2	@0	@3

Figure 4.17: Mouvement *NewSwapHomes*

T\R	**0**	1	2	**3**	4	5	6	7	8	9	10	11	12	13
0	**@1**	2	3	**5**	@6	@3	**@5**	**1**	4	6	@7	@4	7	@2
1	**0**	@5	2	@3	7	@2	@4	**@0**	3	@7	@6	5	6	4
2	3	@0	@1	@7	5	1	6	@3	@6	4	@5	7	@4	0
3	@2	@6	@0	1	4	0	@7	2	@1	5	@4	6	@5	7
4	5	7	@5	6	@3	@6	1	@7	@0	@2	3	0	2	@1
5	@4	1	4	**@0**	@2	@7	**0**	6	7	@3	2	@1	3	@6
6	7	3	@7	@4	0	4	@2	@5	2	@0	1	@3	@1	5
7	@6	@4	6	2	@1	5	3	4	@5	1	0	@2	@0	@3

Figure 4.18: Matchs du mouvement *NewSwapHomes*

T\R	**0**	1	2	**3**	4	5	6	7	8	9	10	11	12	13
0	**1**	2	3	**@5**	@6	@3	**5**	**@1**	4	6	@7	@4	7	@2
1	**@0**	@5	2	@3	7	@2	@4	**0**	3	@7	@6	5	6	4
2	3	@0	@1	@7	5	1	6	@3	@6	4	@5	7	@4	0
3	@2	@6	@0	1	4	0	@7	2	@1	5	@4	6	@5	7
4	5	7	@5	6	@3	@6	1	@7	@0	@2	3	0	2	@1
5	@4	1	4	**0**	@2	@7	**@0**	6	7	@3	2	@1	3	@6
6	7	3	@7	@4	0	4	@2	@5	2	@0	1	@3	@1	5
7	@6	@4	6	2	@1	5	3	4	@5	1	0	@2	@0	@3

Figure 4.19: État final du mouvement *NewSwapHomes*

La taille du voisinage V pour une itération est le nombre d'équipes n multiplié par l'ensemble des combinaisons possibles de $R_1 < R_2$. Soit κ le nombre de rondes ($2n - 2$ pour le TTP), la formule (4.58) permet de calculer la taille du voisinage pour le TTP.

$$|V| = n \cdot \frac{\kappa(\kappa - 1)}{2} \tag{4.58}$$

Dans le cas du BSSP, pour effectuer un mouvement *SwapHomes* avec la modélisation du voisinage *NewSwapHomes*, nous posons la contrainte $R_2 = R_1 + 23$. La variable R_1 sera toujours dans la première moitié de l'horaire et la variable R_2 à la même ronde de la seconde moitié de l'horaire. En posant cette contrainte, nous assurons de toujours choisir deux matchs contre le même adversaire comme le veut le mouvement *SwapHomes*. La taille du voisinage dans ce cas est simplement le nombre d'équipes multiplié par la moitié de l'horaire ($\kappa = 23$), formule (4.59).

$$|V| = n \cdot \kappa \tag{4.59}$$

Contraintes d'interface

Nous exprimons les contraintes de ce voisinage selon la nouvelle forme. Les traitements que nous offrons pour toutes ces contraintes sont cependant activés lorsque toutes les variables de voisinage impliquées sont affectées.

La contrainte (4.60) change les lieux des matchs pour l'horaire de l'équipe T, alors que la contrainte (4.61) ajuste les horaires des adversaires de T dans les rondes R_1 et R_2 ($ad[\sigma_{T,R_1}]$ et $ad[\sigma_{T,R_2}]$). Ces deux contraintes utilisent l'adversaire actuel du match à modifier et inverse le lieu avec l'expression $l[\sigma_{T,R}]$. Les autres matchs de l'horaire gardent leurs valeurs de la solution courante. La contrainte (4.62) fixe les matchs qui ne changent pas dans l'horaire de l'équipe T. La contrainte (4.63) fait la même chose pour l'horaire des adversaires de T dans les rondes R_1 et R_2.

Finalement la contrainte (4.64) s'occupe de toutes les autres équipes.

$$\bigwedge_{\forall j | ad[\sigma_{T,j}] \in \{ad[\sigma_{T,R_1}], ad[\sigma_{T,R_2}]\}} M_{T,j} = ad[\sigma_{T,j}] + (1 - l[\sigma_{T,j}]) \cdot n \tag{4.60}$$

$$\bigwedge_{\forall j | ad[\sigma_{T,j}] \in \{ad[\sigma_{T,R_1}], ad[\sigma_{T,R_2}]\}} M_{ad[\sigma_{T,j}],j} = T + l[\sigma_{T,j}] \cdot n \tag{4.61}$$

$$\bigwedge_{\forall j | ad[\sigma_{t,j}] \notin \{ad[\sigma_{T,R_1}], ad[\sigma_{T,R_2}]\}} M_{T,j} = \sigma_{T,j} \tag{4.62}$$

$$\bigwedge_{\forall j | ad[\sigma_{T,j}] \notin \{ad[\sigma_{T,R_1}], ad[\sigma_{T,R_2}]\}} M_{ad[\sigma_{T,j}],j} = \sigma_{ad[\sigma_{T,j}],j} \tag{4.63}$$

$$\bigwedge_{\forall i \notin \{t, ad[\sigma_{T,R_1}]), ad[\sigma_{T,R_2}]\} \forall j} M_{i,j} = \sigma_{i,j} \tag{4.64}$$

4.3.2 Recherche à grand voisinage

Comme mentionné précédemment, la recherche à grand voisinage est utilisée lorsque tous les mouvements de recherche locale ont atteint un optimum local. L'idée est de préserver une portion de la solution courante et d'utiliser le modèle de programmation par contraintes pour chercher de nouvelles solutions de meilleur coût.

L'objectif de ce type de voisinage est de chercher à préserver les bonnes décisions prises dans la solution pour ainsi diriger rapidement la recherche dans les zones les plus prometteuses de l'espace de recherche. Les décisions sont en fait les variables de décision du modèle. On doit choisir le nombre de variables qui seront fixées. Selon ce nombre de variables, il y aura un certain nombre de préaffectations possibles. Si n est le nombre de variables de décision et m le nombre de variables qui sont fixées, la formule $n!/(m! \cdot (n-m))$ permet de calculer le nombre de préaffectations

possibles. Lorsque le nombre de variables fixées est petit, il y a peu de préaffectations possibles, mais l'espace de recherche formé par les autres variables est grand. Lorsque le nombre de variables fixées est grand, il y a encore peu de préaffectations possibles, mais l'espace de recherche formé par les autres variables est alors très petit. C'est lorsque que $m = n/2$ qu'il y a le plus de préaffectations possibles, la taille de l'espace de recherche formé par les autres variables est alors un meilleur compromis entre les deux extrêmes. Il faut donc doser la proportion des variables qui seront fixées pour obtenir un voisinage qu'on peut explorer efficacement.

Pour certains problèmes, il est cependant difficile de déterminer ce qui est une bonne décision, car chaque décision a des répercussions sur d'autres décisions. Ainsi pour optimiser globalement, il faut dans certains cas délaisser des décisions, qui localement, sont excellentes. Le TTP et le BSSP sont de bons exemples de ce type de problèmes. Nous avons ainsi tenté de déterminer par différentes heuristiques les bonnes décisions, mais sans réel succès. Cette approche n'a donc pas été retenue, mais les principales pistes explorées sont présentées dans les paragraphes qui suivent; d'abord des tentatives pour préserver les meilleures tournées et ensuite, pour préserver les meilleurs voyages.

Meilleures tournées

On peut en quelque sorte considérer les horaires des équipes comme plusieurs tournées. Le voisinage consiste alors à préserver un certain nombre des meilleures tournées de l'horaire. Intuitivement, les tournées les plus courtes semblent être les meilleures, mais il faut tenir compte de l'éloignement de certaines équipes. Dans la NBL, il y a plus d'équipes dans le nord-est des États-Unis qu'en Californie. Il est donc normal que les équipes de la Californie aient des tournées qui sont plus longues puisque les distances aux autres équipes sont en moyenne plus grandes. Il

faut donc normaliser les longueurs des tournées afin de déterminer lesquelles sont réellement bonnes. Pour cela, nous avons utilisé la somme des distances d'une équipe à toutes les autres. Ainsi, si T_i est la longueur de la tournée de l'équipe i et que d_{ij} est la distance de l'équipe i à j alors l'équation (4.65) détermine la valeur de τ_i, la longueur normalisée de l'équipe i. Cette mesure permet alors d'identifier les équipes dont la tournée sera gardée.

$$\tau_i = \frac{T_i}{2 \cdot \sum_{j=1}^{n} d_{ij}} \tag{4.65}$$

Deux options sont alors possibles pour le voisinage. La première option consiste à fixer la tournée de très peu d'équipes de l'horaire. Par exemple, uniquement les trois meilleures tournées d'un horaire pour douze équipes. Ensuite, chercher uniquement dans cet espace de recherche de nouvelles solutions. La seconde option consiste à fixer l'horaire d'un grand nombre d'équipes, cela mène à un espace de recherche plus rapide à explorer complètement, puisqu'on permet alors de fixer différentes combinaisons de tournées. Par exemple, fixer six des neuf meilleures tournées dans un horaire de douze équipes donne $9!/(6! \cdot (9-6)!) = 84$ combinaisons. En jouant avec les paramètres, on peut contrôler le nombre de combinaisons et la taille de l'espace de recherche. La première option ne s'est pas avérée efficace, car nous n'avons pas de bonne heuristique pour la recherche de solutions dans espace de recherche de grande taille. La seconde option s'est montrée plus efficace, mais uniquement lorsque le nombre de combinaisons explorées est très élevé. Pour cela, il faut pouvoir fixer les tournées de toutes les équipes. Ainsi, garder uniquement des tournées parmi les meilleures tournées ne s'est pas montré utile.

Une autre idée est de forcer l'amélioration de la pire tournée avant de compléter l'horaire. Pour cela, quelques bonnes tournées sont fixées pour réduire l'espace de recherche. Ensuite un modèle pour une seule équipe est utilisé pour remplacer la pire tournée de l'horaire. Les nouvelles tournées proposées doivent respecter la

portion de l'horaire déjà fixée. Cette idée n'a cependant pas été complètement explorée faute de temps. Le modèle ne générait qu'une seule nouvelle tournée pour la pire équipe qui était ensuite utilisée pour une recherche. Les résultats étaient décevants. Il aurait été préférable à notre avis de générer plusieurs nouvelles tournées.

Meilleurs voyages

Dans la tournée d'une équipe, uniquement les matchs sur la route contribuent à sa longueur. Nous avons donc exploré l'idée de préserver uniquement les bons voyages de deux ou trois matchs. La longueur des voyages est normalisée comme les tournées pour ne pas pénaliser les équipes éloignées. Une première approche consiste à fixer une portion des bons voyages dans l'horaire. Cependant, même en fixant tous les voyages de deux ou trois matchs l'espace de recherche est trop grand.

Une seconde approche consiste à utiliser un modèle qui permet de déplacer les voyages dans les tournées. L'idée est que nous voulons garder les voyages, mais pas nécessairement leur emplacement dans les tournées. Le modèle permet ainsi de tester toutes les configurations possibles des bons voyages dans l'horaire. Le modèle s'est montré trop difficile à résoudre pour tous les voyages; nous avons du restreindre le nombre de voyages pouvant être déplacés. Nous avons cependant eu le même problème que pour la première approche; l'espace de recherche reste trop grand.

Pour les deux approches, il faudrait fixer d'autres matchs dans l'horaire de façon à réduire l'espace de recherche. Ce sont cependant des expérimentations que nous n'avons pas eu le temps de faire.

Approche proposée

De toutes les expérimentations que nous avons effectuées, celle qui s'est montrée la plus efficace et la plus robuste est l'approche consistant à fixer un sous-ensemble des meilleures tournées. Un problème avec cette approche, c'est qu'en considérant uniquement les meilleures tournées le voisinage est restreint à quelques préaffectations possibles. Pour augmenter la taille du voisinage, deux options sont possibles : fixer moins de tournées et permettre de fixer des tournées moins bonnes. C'est ce que nous proposons dans notre approche.

Nous proposons donc une version systématique de cette approche basée sur les meilleures tournées. Dans notre approche, toutes les tournées peuvent être fixées pour obtenir beaucoup plus de préaffectations possibles. Selon la taille du problème, m est le nombre de tournées qui doivent être préaffectées dans l'horaire. L'idée est d'essayer tous les sous-ensembles possibles, pour ainsi obtenir plus de préaffectations possibles. Il y a par exemple $16!/(10! \cdot (16-10)!) = 8008$ sous-ensembles de dix équipes dans seize équipes. Comme les heuristiques de recherche ne sont pas très efficaces, nous avons choisi m pour que l'espace de recherche formé par les autres variables puisse être exploré soit explorable en quelques secondes. Accorder une seconde d'exploration à chacune de ces 8008 préaffectations donne déjà plus de deux heures de recherche.

La recherche à grand voisinage comporte deux phases. La première phase explore les préaffectations de m tournées jusqu'à l'obtention d'une nouvelle solution meilleure que la solution courante. La recherche arrête dès qu'une meilleure solution est trouvée. La seconde phase est une répétition de la première, mais en réduisant le nombre de tournées fixées à $m-1$. Cette phase est utilisée uniquement lorsque la première n'est pas fructueuse. Les voisinages sont trop grands pour l'efficacité des heuristiques de recherche, mais c'est le dernier recours pour utiliser le temps

accordé pour optimiser le problème. Pour que la recherche ne perde pas tout son temps dans les premières mauvaises préaffectations, nous avons décidé de limiter le temps d'exploration de chacune des préaffectations. Le chapitre qui suit présente en détail les expérimentations et les résultats de ce projet.

CHAPITRE 5

RÉSULTATS

Ce chapitre d'expérimentations et de résultats se divise en trois sections. D'abord, nous présentons les stratégies d'exploration et les résultats des cinq voisinages de recherche locale, ensuite les stratégies et les résultats de la descente à voisinage variable sans le grand voisinage et finalement, la stratégie et les résultats de la descente à voisinage variable avec le grand voisinage. Tout le projet a été développé en C++/Ilog Solver 6.0 [18, 17] sur une plate forme Linux. Les tests ont été exécutés sur des machines avec processeur Intel Pentium 4 à 3Ghz sur la distribution Fedora Core 1.

5.1 Voisinages de recherche locale

Dans cette section, nous testons individuellement chacun des voisinages de recherche locale. Nous voulons déterminer les meilleures stratégies d'exploration et la capacité à optimiser les problèmes. La stratégie de base d'exploration des voisinages consiste à explorer l'arbre de recherche formé par les variables de voisinage. Les contraintes d'interface permettent de définir les nouvelles solutions et de rejeter des voisins qui ne sont pas valides. Le TTP et le BSSP sont des problèmes d'optimisation, nous voulons donc que ces nouvelles solutions soient meilleures que les solutions précédentes. Pour cela, nous maintenons la valeur de la solution courante pendant la recherche locale et nous évaluons si les affectations partielles sont susceptibles de mener à de meilleures solutions. Ainsi, à chaque affectation de variables, nous examinons les combinaisons de valeurs dans les domaines courants

des variables de voisinage pour trouver une affectation complète de meilleur coût que la solution courante. L'exploration du voisinage peut se poursuivre dès qu'une telle affectation complète est trouvée. Cependant, si une telle affectation complète n'existe pas, alors l'affectation partielle est rejetée et nous pouvons élaguer cette branche de l'arbre de recherche afin d'accélérer la recherche. Pour que le processus d'évaluation des combinaisons soit rapide, nous ne vérifions pas explicitement les contraintes du problème. Cependant, la propagation et le filtrage des domaines en cours de recherche aide à éliminer des valeurs dans les domaines qui ne sont pas cohérentes avec les affectations. Cela réduit le nombre de combinaisons à évaluer. De plus, nous gardons en mémoire l'ancienne meilleure combinaison comme raccourci. Si cette combinaison est encore dans le domaine des variables c'est qu'elle est encore valide. Il n'est donc pas nécessaire de chercher une nouvelle meilleure combinaison. Nous utilisons cette technique dans toutes les stratégies d'exploration des voisinages.

Nous utiliserons l'expression *stratégie de base* pour représenter l'ordre d'affectation des variables et l'ordre de choix des valeurs. Pour tous les voisinages, nous allons comparer quelques stratégies de base. D'abord, la stratégie de base la plus courante qui consiste à affecter en premier les variables ayant le plus petit domaine et de choisir les valeurs en ordre lexicographique. Ensuite, des stratégies qui profitent des classes d'équivalence. Pour chaque voisinage, nous allons identifier ces stratégies de base par des lettres; la stratégie du plus petit domaine d'abord, que nous utiliserons pour tous les voisinages, sera toujours la stratégie de base A.

Pour chacune de ces stratégies de base, nous allons aussi tester l'impact de la fragmentation de domaine. La fragmentation de domaine permet une stratégie de recherche qui ne restreint pas trop rapidement le domaine des variables. Plutôt que d'affecter une valeur à une variable dès qu'elle est choisie, le domaine de cette variable est séparé en deux sous-domaines. Comme mentionné à la section 1.4.3,

cette technique est généralement intéressante pour des variables ayant de grands domaines et lorsqu'il y a une bonne propagation malgré une moins grande réduction de domaine, ce qui est le cas de nos voisinages. La fragmentation de domaine nous permettra aussi d'évaluer les combinaisons de valeurs des domaines plus souvent. Nous allons faire cette évaluation à chaque fois que le domaine d'une variable est séparé; ainsi, il sera aussi possible de rejeter un sous-domaine complet s'il ne contient aucune meilleure combinaison de valeurs. Cela devrait permettre de couper une plus grande portion de l'arbre et ainsi accélérer l'exploration même si l'évaluation des combinaisons se fera plus souvent. Le critère de séparation que nous avons utilisé est la moyenne des bornes du domaine, soit pour une variable x : $(min(D(x)) + max(D(x)))/2$. Pour les mouvements *PartialSwapTwoRounds* et *PartialSwapThreeRounds*, nous allons aussi tester l'impact du réveil de contraintes sur l'événement *WhenRange*.

La méthodologie que nous avons choisie pour les expérimentations consiste à faire plusieurs descentes de recherche locale avec le voisinage testé. Chacune de ces descentes utilise une solution de départ différente; nous avons trouvé ces solutions en utilisant les modèles de programmation par contraintes des problèmes présentés au début du chapitre de mise en oeuvre. Nous avons fixé la première variable de ce modèle à des valeurs différentes pour assurer la diversité de ces solutions initiales. Les résultats présentés sont les moyennes de toutes les descentes. Pour identifier complètement nos stratégies, nous adoptons une convention qui combine les trois paramètres suivants : stratégie de base[A,B,C,D,E], fragmentation de domaine[x=non, F=oui] et événement de réveil [V=*WhenValue*, R=*WhenRange*]. Ainsi la stratégie «A_x_V» est basée sur la stratégie de base A du voisinage considéré, n'utilise pas la fragmentation de domaine et les contraintes sont toutes réveillées sur l'événement *WhenValue*. Nous allons présenter dans des tableaux comparatifs les résultats de chacune des stratégies pour chaque voisinage, c'est-à-dire le

nombre d'échecs, le pourcentage du voisinage exploré, le temps d'exploration et le nombre d'itérations. Nous allons expliquer en détail chacune de ces mesures pour le voisinage *PartialSwapTwoRounds*; il faudra donc s'y référer pour les autres voisinages.

Dans un autre tableau, nous allons finalement montrer dans quelle mesure les voisinages sont capables d'optimiser individuellement les problèmes. Dans ce tableau, nous présenterons essentiellement la valeur moyenne : des solutions initiales, des solutions obtenues suite aux descentes de recherche locale et le pourcentage d'amélioration.

5.1.1 PartialSwapTwoRounds

Le mouvement *PartialSwapTwoRounds* est représenté par les trois variables T, R_1 et R_2. Nous avons vu dans la description de ce voisinage au chapitre 4, qu'il existe des classes d'équivalence qui permettent de filtrer les valeurs de la variable d'équipe T. Pour profiter de ces classes d'équivalence, il faut connaître les rondes choisies avant de connaître l'équipe. Pour cela, il faut affecter les variables des rondes avant la variable d'équipe. Nous proposons de tester quatre différentes stratégies d'ordre d'affectation et de choix de valeur :

Stratégie	Description
A	plus petit domaine d'abord
B	$R_1 \rightarrow R_2 \rightarrow T$
C	$R_2 \rightarrow R_1 \rightarrow T$
D	$R_{1max} \rightarrow R_2 \rightarrow T$

Toutes ces stratégies de base choisissent les valeurs selon l'ordre lexicographique sauf une exception : la variable R_1 de la stratégie D pour laquelle la plus grande valeur est choisie en premier (R_{1max}).

Tableau 5.1: Résultats de l'exploration du mouvement *PartialSwapTwoRounds* pour le TTP complet (moyenne de 36 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	177969.2	75.02	391.29	11.48
A_F_V	83852.1	35.35	161.58	4.74
B_x_V	18551.2	7.82	66.34	1.95
B_F_V	12724.3	5.36	35.36	1.04
C_x_V	18283.4	7.71	64.41	1.89
C_F_V	12384.6	5.22	34.24	1.00
D_x_V	17414.3	7.34	60.46	1.77
D_F_V	11731.9	4.95	30.93	0.91
A_x_R	177969.2	75.02	358.98	10.53
A_F_R	83852.1	35.35	268.92	7.89
B_x_R	18551.2	7.82	51.62	1.51
B_F_R	12724.3	5.36	27.72	0.81
C_x_R	18283.4	7.71	57.06	1.67
C_F_R	12384.6	5.22	26.67	0.78
D_x_R	17414.3	7.34	53.64	1.57
D_F_R	11731.9	4.95	25.51	0.75

Les stratégies B, C et D sont conçues pour s'assurer d'affecter les variables de rondes en premier. Nous proposons les stratégies C et D, car elles sont différentes de la stratégie B. En effet, lorsque la variable R_2 est affectée en premier, les premières valeurs choisies seront les premières rondes du tournoi. Nous savons que R_1 doit être inférieure à la variable R_2, cela fait en sorte que le domaine de R_1 sera limité à un très petit nombre de valeurs en début de recherche. De la même façon, si R_1 est affectée en premier en choisissant les dernières rondes du tournoi, le domaine de R_2 sera aussi réduit à un très petit nombre de valeurs en début de recherche. On voit donc que les stratégies C et D sont en fait très similaires. Nous pensons donc qu'elles devraient obtenir des résultats relativement semblables.

Les tableaux 5.1 et 5.2 présentent les résultats moyens de l'exploration du voisinage avec chacune des stratégies. La première colonne indique la stratégie choisie. La

Tableau 5.2: Résultats de l'exploration du mouvement *PartialSwapTwoRounds* pour le BSSP (moyenne de 25 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	4406.2	6.74	50.08	4.65
A_F_V	4300.0	6.58	48.54	4.51
B_x_V	4406.2	6.74	49.41	4.59
B_F_V	4300.0	6.58	48.11	4.47
C_x_V	4348.3	6.66	49.83	4.63
C_F_V	4265.7	6.53	48.87	4.54
D_x_V	4357.2	6.67	50.18	4.66
D_F_V	4232.3	6.48	47.80	4.44
A_x_R	4406.2	6.74	36.97	3.44
A_F_R	4300.0	6.58	27.03	2.51
B_x_R	4406.2	6.74	37.08	3.45
B_F_R	4300.0	6.58	27.10	2.52
C_x_R	4348.3	6.66	48.46	4.50
C_F_R	4265.7	6.53	26.34	2.45
D_x_R	4357.2	6.67	44.51	4.14
D_F_R	4232.3	6.48	27.01	2.51

colonne *Échecs* donne le nombre d'échecs effectués par le solveur durant l'exploration du voisinage. Les échecs sont les noeuds de l'arbre qui ont menés à une affectation partielle (ou complète) non valide. Au nombre d'échecs, nous ajoutons aussi le nombre de feuilles de l'arbre qui n'ont pas créées d'échecs. Le nombre d'échecs est une mesure donnant un ordre de grandeur du nombre de noeuds explorés dans l'arbre complet. Pour évaluer le portion de l'arbre de recherche exploré, nous utilisons la mesure ξ qui est le nombre d'échecs divisé par la taille du voisinage. Les trois dernières colonnes donnent le nombre d'itérations effectuées par la descente, le temps total et le temps par itération. Pour calculer ξ, nous utilisons la formule (4.16) qui donne la taille du voisinage : soient 6960 pour le TTP et 6072 pour le BSSP que nous devons ensuite multiplier par le nombre d'itérations. L'objectif est de trouver la meilleure stratégie, c'est-à-dire celle qui a un petit ξ et un temps d'exploration rapide. Toutes ces stratégies trouvent le meilleur voisin à chaque itération, la descente est donc la même pour chacune de ces stratégies. Nous ne présentons donc pas les valeurs des solutions initiales et solutions obtenues car elles sont les mêmes pour toutes les stratégies. Nous allons plutôt présenter ces résultats pour chacun des voisinages à la fin de cette section.

En étudiant les résultats du tableau 5.1, on remarque rapidement que la stratégie de base A est beaucoup moins efficace que les trois autres. Cette stratégie affecte en premier la variable d'équipe; dans ce cas, il est impossible d'utiliser les classes d'équivalence. Une grande portion de l'espace de recherche est explorée, jusqu'à 75% lorsqu'il n'y a pas de fragmentation de domaine. On remarque ensuite que la fragmentation a une influence importante dans l'exploration de toutes les stratégies; le nombre d'échecs est significativement réduit et cela se traduit par des temps d'exploration qui sont environ deux fois plus rapides. Réveiller les contraintes sur l'événement *WhenRange* ne change pas le nombre d'échecs, cependant les temps de calcul sont améliorés d'environ 10% pour les stratégies A, B et C,

Tableau 5.3: Réveils des contraintes du mouvement *PartialSwapTwoRounds* pour le TTP complet (moyenne de 36 explorations de 1 itération)

Stratégie	Échecs	Réveils	Évaluations	Temps (s)
A_x_V	4585.1	6860.8	643.0	9.38
A_F_V	2244.3	2257.4	1686.0	4.17
A_x_R	4585.1	9587.1	643.0	8.79
A_F_R	2244.3	4837.0	1686.0	6.82
D_x_V	375.2	638.0	174.0	1.32
D_F_V	236.1	280.2	259.0	0.62
D_x_R	375.2	648.0	174.0	1.21
D_F_R	236.1	374.9	259.0	0.56

mais pour la stratégie A l'exploration est plus lente. Nous ne pouvons expliquer pourquoi l'événement *WhenRange* accélère les explorations, alors que le nombre d'échecs ne varie pas, particulièrement sans la fragmentation de domaine. La piste la plus plausible est que l'implémentation des contraintes pour cet événement est plus efficace pour le solveur. Le tableau 5.3 présente le nombre moyen de réveils des contraintes (4.24), (4.25) et (4.26) pour l'exploration d'une itération du voisinage *PartialSwapTwoRounds*. La colonne *Réveils* est le nombre total de réveils de ces contraintes et la colonne *Évaluation* est le nombre de fois que la fonction évaluant les combinaisons est appelée. Ces résultats permettent de comprendre pourquoi la stratégie A_F_V est plus rapide que la stratégie A_F_R alors que normalement l'événement *WhenRange* accélère la recherche. On voit comme prévu que la fragmentation de domaine réduit le nombre d'échecs et de réveils mais augmente le nombre d'appels à l'évaluation des combinaisons. Ce qui est différent pour la stratégie de base A, c'est que le nombre de réveils de la stratégie _F_R par rapport _F_V est plus élevé que pour la stratégie de base D. En effet, il y a 2.14 fois plus de réveils de la stratégie A_F_R que A_F_V, alors qu'il y a 1.40 fois plus de réveils de la stratégie D_F_R que D_F_V.

En étudiant les résultats du tableau 5.2, on remarque que les stratégies sont plus égales pour le BSSP que pour le TTP. Lorsque les contraintes sont réveillées sur l'événement $WhenValue$, seule la fragmentation de domaine permet de réduire un peu le nombre d'échecs et cela de façon constante pour les stratégies de base A, B, C et D. Réveiller les contraintes sur l'événement $WhenRange$ a un plus grand impact sur le BSSP que le TTP; combinée à la fragmentation de domaine, l'exploration de toutes les stratégies est accélérée d'environ 44% et elles sont presque toutes égales. Nous sommes surpris que la stratégie A ait obtenu des résultats similaires aux stratégies B, C et D qui profitent des classes d'équivalence. Nous attribuons ce comportement à la structure en miroir du BSSP, qui a plus d'équipes et moins de rondes que le TTP.

Suite à ces expérimentations, nous jugeons que la fragmentation de domaine et que le réveil des contraintes sur l'événement $WhenRange$ sont utiles pour l'exploration de ce voisinage. La stratégie D_F_R s'est montrée la plus rapide pour le TTP, alors que c'est la stratégie C_F_R qui est la plus rapide pour le BSSP. Pour les deux problèmes, nous remarquons que la stratégie D est celle qui présente le plus petit nombre d'échecs. Nous avons cependant décidé d'utiliser une seule stratégie pour l'exploration de ce voisinage. Nous optons pour la stratégie D_F_R, qui a le moins d'échecs pour les deux problèmes et qui est la plus rapide pour le TTP.

5.1.2 PartialSwapTwoTeams

Le mouvement $PartialSwapTwoTeams$ est représenté par les trois variables T_a, T_b et R. Nous avons vu dans la description de ce voisinage au chapitre 4 qu'il existe des classes d'équivalence qui permettent de filtrer les valeurs de la variable de rondes R. Pour profiter de ces classes d'équivalence, il faut connaître les équipes choisies avant de connaître la ronde. Pour cela, il faut affecter les variables des

équipes avant la variable de ronde. Nous proposons de tester quatre différentes stratégies d'ordre d'affectation et de choix de valeurs :

Stratégie	Description
A	plus petit domaine d'abord
B	$T_a \rightarrow T_b \rightarrow R$
C	$T_b \rightarrow T_a \rightarrow R$
D	$T_a max \rightarrow T_b \rightarrow R$
E	$R \rightarrow$ plus petit domaine entre T_a et T_b

Toutes ces stratégies de base choisissent les valeurs selon l'ordre lexicographique sauf une exception : la variable T_a de la stratégie D pour laquelle la plus grande valeur qui est choisie en premier ($T_a max$).

Les stratégies B, C et D sont essentiellement les mêmes que pour le mouvement *PartialSwapTwoRounds*, nous avons seulement inversé les variables de rondes et les variables d'équipes afin de profiter des classes d'équivalence.

Si les domaines des variables d'équipes sont plus petits que celui de la variable de ronde, alors la stratégie A profite aussi des classes d'équivalence. C'est le cas du TTP, puisque la variable R a un domaine presque deux fois plus grand que les variables d'équipes. Pour le BSSP, les domaines des variables sont de taille égale au début de l'exploration. En effet, comme $T_a < T_b$ et qu'il y a 24 équipes, ces deux variables ont 23 valeurs dans leur domaine. La variable de ronde a comme domaine la première moitié du tournoi, soit 23 valeurs. Le solveur choisit alors les variables dans l'ordre qu'elles lui ont été présentées, dans notre implémentation ce sont les variables d'équipes. La stratégie A devrait donc être similaire aux stratégies B, C et D. Pour cette raison, nous proposons de tester la stratégie E qui affecte en premier la variable R et choisi ensuite la variable d'équipes ayant le plus petit domaine. Cette stratégie sert à vérifier si les classes d'équivalence sont vraiment utiles.

Tableau 5.4: Résultats de l'exploration du mouvement *PartialSwapTwoTeams* pour le TTP complet (moyenne de 36 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	5007.2	5.16	15.92	0.59
A_F_V	3671.6	3.78	11.37	0.42
B_x_V	5007.2	5.16	15.92	0.59
B_F_V	3671.6	3.78	11.43	0.42
C_x_V	5140.3	5.29	16.45	0.61
C_F_V	3716.6	3.83	11.69	0.43
D_x_V	5007.2	5.16	15.89	0.59
D_F_V	3613.8	3.72	11.24	0.42
E_x_V	13967.7	14.38	42.37	1.57
E_F_V	6279.9	6.47	13.50	0.50

Tableau 5.5: Résultats de l'exploration du mouvement *PartialSwapTwoTeams* pour le BSSP (moyenne de 25 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	20716.9	9.42	164.34	4.74
A_F_V	18604.7	8.44	140.47	4.06
B_x_V	20716.9	9.42	166.10	4.79
B_F_V	18604.7	8.44	142.67	4.12
C_x_V	20918.4	9.49	169.15	4.88
C_F_V	18744.6	8.50	143.96	4.16
D_x_V	20716.9	9.42	164.06	4.74
D_F_V	18422.8	8.38	139.12	4.02
E_x_V	60362.3	32.79	629.04	21.69
E_F_V	30608.7	15.55	210.99	6.81

Les tableaux 5.4 et 5.5 présentent les résultats moyens de l'exploration du voisinage *PartialSwapTwoTeams* avec chacune des stratégies. Toutes les stratégies de ce voisinage utilisent uniquement l'événement $WhenValue$, car il n'y a pas de contrainte pouvant être réveillée sur l'événement $WhenRange$. On remarque, comme prévu, que la stratégie du plus petit domaine d'abord présente des résultats similaires à ceux des stratégies B, C et D. En regardant les nombres d'échecs des stratégies A et B, on peut confirmer que la stratégie du plus petit domaine d'abord est équivalente à la stratégie B pour le TTP et le BSSP. La stratégie A serait cependant différente pour un problème qui aurait plus d'équipes que de rondes. La stratégie E_x_V montre que les classes d'équivalence sont très utiles, les nombres d'échecs et les temps d'exécution sont beaucoup plus grands.

La fragmentation de domaine accélère l'exploration des stratégies A, B, C et D d'environ 29% pour le TTP et d'environ 16% pour le BSSP. L'accélération pour la stratégie E_F_V est environ 68%. Pour calculer ξ, nous utilisons la formule (4.30) qui donne la taille du voisinage : soient 3690 pour le TTP et 97092 pour le BSSP que nous devons ensuite multiplier par le nombre d'itérations. Nous choisissons la stratégie D_F_V pour explorer ce voisinage, car elle est légèrement plus efficace que les autres, en terme de temps et de nombre d'échecs.

5.1.3 PartialSwapThreeRounds

Le mouvement *PartialSwapThreeRounds* est représenté par les cinq variables T_a, T_b, R_1, R_2 et R_3. Il peut profiter des classes d'équivalence pour filtrer les variables d'équipes lorsque les variables de rondes sont déterminées. Ce mouvement est en quelque sorte deux mouvements *PartialSwapTwoRounds* consécutifs. Pour cette raison, nous avons vu au chapitre 4 que ce mouvement est modélisé en deux étapes. Lorsque les variables R_1, R_2 et T_a sont affectées, l'échange entre les deux

Tableau 5.6: Résultats de l'exploration du mouvement *PartialSwapTwoTeams* pour le TTP complet (moyenne de 36 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	1988271.5	15.16	6273.95	233.09
A_F_V	807840.8	6.16	1632.60	60.65
D_x_V	439994.0	3.36	516.54	19.19
D_F_V	217735.0	1.66	223.53	8.30
A_x_R	1988271.5	15.16	3935.11	146.20
A_F_R	807840.8	6.16	2956.96	109.86
D_x_R	439994.0	3.36	403.55	14.99
D_F_R	217735.0	1.66	191.89	7.13

premières rondes peut se faire. Le résultat est préservé en attente des deux autres variables pour procéder au second échange. En suivant cette séquence logique et en réutilisant les résultats obtenus pour le mouvement *PartialSwapTwoRounds*, nous proposons de comparer les deux stratégies de recherche suivantes :

Stratégie	Description
A	plus petit domaine d'abord
D	$R_{1max} \rightarrow R_2 \rightarrow T_a \rightarrow R_3 \rightarrow T_b$

La première stratégie est celle du plus petit domaine d'abord. La seconde stratégie est basée sur les résultats obtenus pour le voisinage *PartialSwapTwoRounds*, nous réutilisons la stratégie choisie à ce moment pour affecter les variables R_1, R_2, T_a et nous la complétons en affectant la variable R_3 avant la variable T_b. L'idée est de s'assurer de pouvoir profiter des classes d'équivalence dans la première phase du mouvement et dans la seconde.

Les tableaux 5.6 et 5.7 présentent les résultats obtenus. On remarque un patron tout à fait similaire aux résultats du mouvement *PartialSwapTwoRounds*. Pour le TTP, la stratégie D est beaucoup plus efficace que la stratégie A. La fragmentation de domaine réduit au moins de moitié le nombre d'échecs et le temps d'exploration.

Tableau 5.7: Résultats de l'exploration du mouvement *PartialSwapTwoTeams* pour le BSSP (moyenne de 36 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	255631.4	13.34	217.95	55.60
A_F_V	90662.0	4.73	215.29	54.92
D_x_V	244348.2	12.75	218.13	55.65
D_F_V	86199.9	4.50	212.88	54.31
A_x_R	255631.4	13.34	161.47	41.19
A_F_R	90662.0	4.73	117.11	29.88
D_x_R	244348.2	12.75	164.15	41.88
D_F_R	86199.9	4.50	116.53	29.73

Le réveil sur l'événement *WhenRange* mène à des temps d'exploration plus rapides sauf pour la stratégie A_F_R. Pour le BSSP, les stratégies A et D sont plutôt équivalentes, mais on remarque qu'il y a moins d'échecs pour la stratégie D. La fragmentation de domaine réduit de beaucoup le nombre d'échecs ce qui se traduit par une accélération de 28% du temps d'exploration. Pour calculer ξ, nous utilisons la formule (4.36) qui donne la taille du voisinage : soient 487200 pour le TTP et 488796 pour le BSSP que nous devons ensuite multiplier par le nombre d'itérations. Nous utilisons donc la stratégie D_F_R pour explorer ce voisinage.

5.1.4 PartialSwapThreeTeams

Le mouvement *PartialSwapThreeTeams* est représenté par les cinq variables T_a, T_b, T_c, R_1, et R_2. Il peut profiter des classes d'équivalence pour filtrer les variables de rondes lorsque les variables d'équipes sont déterminées. Ce mouvement est en quelque sorte deux mouvements *PartialSwapTwoTeams* consécutifs. Pour cette raison, nous avons vu au chapitre 4 que ce mouvement est modélisé en deux étapes. Lorsque les variables T_a, T_b et R_1 sont affectées, l'échange entre les deux

Tableau 5.8: Résultats de l'exploration du mouvement *PartialSwapThreeTeams* pour le TTP complet (moyenne de 36 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	155875.6	2.72	735.42	31.22
A_F_V	40760.9	0.71	82.40	3.50
D_x_V	158009.2	2.75	773.95	32.86
D_F_V	41389.8	0.72	85.79	3.64

premières équipes peut se faire. Le résultat est préservé en attente des deux autres variables pour procéder au second échange. En suivant cette séquence logique et en réutilisant les résultats obtenus pour le mouvement *PartialSwapTwoTeams*, nous proposons de comparer les deux stratégies de recherche suivantes :

Stratégie	Description
A	plus petit domaine d'abord
D	$T_amax \rightarrow T_b \rightarrow R_1 \rightarrow T_c \rightarrow R_2$

La première stratégie est celle du plus petit domaine d'abord. La seconde stratégie est basée sur les résultats obtenus pour le voisinage *PartialSwapTwoTeams*, nous réutilisons la stratégie choisie à ce moment pour explorer les variables T_a, T_b, R_1 et nous la complétons en affectant la variable T_c avant la variable R_2. L'idée est de s'assurer de pouvoir profiter des classes d'équivalence dans la première phase du mouvement et dans la seconde.

Les tableaux 5.8 et 5.9 présentent les résultats obtenus. On remarque que les résultats obtenus par les deux stratégies sont assez similaires. Cela s'explique par le fait que les variables d'équipes ont un plus petit domaine et sont choisies en premier par la stratégie A. Les classes d'équivalence peuvent être utilisées et les résultats montrent qu'il n'est pas néfaste de choisir T_c avant R_2. Cette stratégie ne pourrait cependant pas en faire autant sur un problème ayant plus d'équipes

Tableau 5.9: Résultats de l'exploration du mouvement *PartialSwapThreeTeams* pour le BSSP (moyenne de 7 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	2224333.6	12.28	24422.95	690.40
A_F_V	719365.4	3.97	4476.50	126.54
D_x_V	2220202.0	12.26	24299.09	686.90
D_F_V	717059.6	3.96	4543.19	128.43

que de rondes. On remarque que la fragmentation de domaine est très utile, le nombre d'échecs et les temps d'exploration sont considérablement réduits. Pour calculer ξ, nous utilisons la formule (4.36) qui donne la taille du voisinage : soient 243600 pour le TTP et 512072 pour le BSSP que nous devons ensuite multiplier par le nombre d'itérations. Nous avons choisi la stratégie D_F_V pour s'assurer de pouvoir toujours profiter des classes d'équivalence et parce que les résultats des stratégies sont à peu près équivalents.

5.1.5 NewSwapHomes

Le mouvement *NewSwapHomes* est représenté par les trois variables T, R_1, et R_2. Comme pour les voisinages précédents, nous proposons de tester la stratégie D, car en début d'exploration le domaine de R_2 est très réduit. Pour le BSSP, nous devons utiliser le voisinage *SwapHomes*. Pour cela, nous utilisons l'implémentation du voisinage *NewSwapHomes* et les même stratégies de recherche, mais R_1 et R_2 sont dépendantes. Le voisinage *SwapHomes* est donc moins grand.

Stratégie	Description
A	plus petit domaine d'abord
D	$R_{1max} \rightarrow R_2 \rightarrow T$

Tableau 5.10: Résultats de l'exploration du mouvement *NewSwapHome* pour le TTP complet (moyenne de 36 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	87431.3	77.70	268.67	16.62
A_F_V	86722.0	77.07	268.97	16.64
D_x_V	111749.5	99.32	353.32	21.85
D_F_V	73324.6	65.17	167.88	10.38

Tableau 5.11: Résultats de l'exploration du mouvement *SwapHome* pour le BSSP (moyenne de 25 exécutions)

Stratégie	Échecs	ξ (%)	T(s)	T/Itér.(s)
A_x_V	1722.2	100.00	25.93	8.31
A_F_V	1544.6	89.68	21.07	6.75
D_x_V	1680.8	97.60	25.36	8.13
D_F_V	1468.8	85.28	19.93	6.39

Les tableaux 5.10 et 5.11 présentent les résultats obtenus. On observe que ξ est très élevé. Pour le voisinage *NewSwapHomes*, une bonne portion des échecs est dû au fait que nous voulons des rondes où un match est sur la route et l'autre à domicile. Nous avons posé $R1 < R_2$ pour éliminer la symétrie qui permuterait deux fois les mêmes deux matchs M_{T,R_1} avec M_{T,R_2}. Cependant, on sait que si on change les domiciles des matchs $M_{T,R_1} = 1$ et $M_{T,R_2} = @5$ on devra aussi permuter les domiciles des deux autres matchs impliquant ces mêmes adversaires, soit pour la même équipe T les matchs $M_{T,R_3} = @1$ et $M_{T,R_4} = 5$. Nous n'avons cependant pas eu le temps de trouver une méthode pour profiter de cette classe d'équivalence. L'exploration du voisinage *NewSwapHomes* pourrait être améliorée en remédiant à ce problème. Pour le BSSP, cette classe d'équivalence n'est pas présente, mais les contraintes d'interface de ce voisinage sont propagées uniquement lorsque toutes les variables sont affectées. Pour cette raison, la stratégie A explore complètement

le voisinage. La stratégie D combinée à la fragmentation de domaine est la plus efficace pour explorer les deux voisinages. Pour calculer ξ, nous utilisons la formule (4.58) qui donne la taille du voisinage pour le TTP et la formule (4.59) pour le BSSP : soient 6960 pour le TTP et 552 pour le BSSP que nous devons ensuite multiplier par le nombre d'itérations.

5.1.6 Descentes

La section précédente présente les différentes stratégies pour explorer les voisinages. Le choix de la stratégie est important, car elle influence grandement la topologie de l'arbre de recherche et les temps d'exploration. Cependant, toutes les stratégies proposées sont équivalentes en ce qui concerne l'optimisation des problèmes. En effet, nous utilisons un modèle de programmation par contraintes pour explorer complètement le voisinage et choisir le meilleur voisin. Ainsi toutes les stratégies proposées, pour un même voisinage, suivent les mêmes chemins durant les descentes et obtiennent les mêmes solutions améliorées. Ce qui diffère, c'est le nombre de mauvaises branches de l'arbre de recherche qui sont élaguées et par le fait même le temps d'exploration du voisinage. Nous présentons dans cette section les résultats des descentes de recherche locale qui sont indépendants des stratégies de recherche : la valeur moyenne des solutions initiales, la valeur moyenne des solutions obtenues, le pourcentage moyen d'amélioration, la pire solution trouvée, la meilleure solution trouvée et le nombre d'itérations. Nous présentons ces résultats dans les tableaux 5.12 et 5.13. L'amélioration des solutions initiales du TTP varie de 7.63% à 15.86%. On remarque pour chaque voisinage, sauf le voisinage $NewSwapHome$, une augmentation du pourcentage d'amélioration en fonction de la taille du TTP. On voit aussi que les grands voisinages $S3R$ et $S3T$ sont meilleurs que les plus petits voisinages $S2R$ et $S2T$. Les voisinages d'échanges entre les rondes sont plus performants, parce que dans le TTP il y a beaucoup de rondes et

Tableau 5.12: Résultats de descente des voisinages pour le TTP (moyenne de 36 exécutions)

n	Type	Sol. Ini.	Rés. Moy.	Amé.%	Pire	Meilleur	Iter
10	S2R	87149.5	76746.0	11.94	81829	71838	14.6
12	S2R	162074.4	143673.7	11.35	150474	136446	20.0
14	S2R	305741.7	266305.0	12.90	280102	249623	29.2
16	S2R	432762.2	372938.5	13.82	398248	353940	34.1
10	S2T	87149.5	79625.2	8.63	84332	74250	9.7
12	S2T	162074.4	147690.7	8.87	154960	137464	13.6
14	S2T	305741.7	273790.0	10.45	296832	255501	21.0
16	S2T	432762.2	383026.1	11.49	402683	357191	27.0
10	S3R	87149.5	76657.7	12.04	81600	71795	10.6
12	S3R	162074.4	142162.8	12.29	149593	132323	15.4
14	S3R	305741.7	260434.4	14.82	279913	240153	22.0
16	S3R	432762.2	364111.7	15.86	380633	343729	26.9
10	S3T	87149.5	78493.1	9.93	83335	74770	8.0
12	S3T	162074.4	144882.9	10.61	154815	136334	12.3
14	S3T	305741.7	265280.3	13.23	282816	247998	18.7
16	S3T	432762.2	371726.9	14.10	386642	342147	23.6
10	NSH	87149.5	79665.5	8.59	83879	75675	8.5
12	NSH	162074.4	149713.6	7.63	155717	143837	10.5
14	NSH	305741.7	281568.8	7.91	295113	269184	13.9
16	NSH	432762.2	396557.6	8.37	407863	380312	16.2

Tableau 5.13: Résultats de descente des voisinages pour le BSSP (moyenne de 25 exécutions)

Type	Sol. Ini.	Rés. Moy.	Amé.%	Pire	Meilleur	Iter
S2R	911221.5	882640.9	3.14	934818	864684	10.8
S2T	911221.5	849978.1	6.72	884663	832553	34.6
S3R	911221.5	893705.7	1.92	931939	873148	3.9
S3T	911221.5	833298.9	9.09	856576	807728	35.4
SH	911221.5	909142.4	0.23	933815	891257	3.1

moins d'équipes. Finalement, on remarque que c'est le voisinage NSH avec 8% d'amélioration, qui est le moins bon pour le TTP. L'amélioration est moins bonne pour les solutions initiales du BSSP, elle varie de 0.23% à 9.09%. Puisqu'il y a peu de rondes et beaucoup d'équipes, ce sont plutôt les échanges entre les équipes qui performent mieux. On remarque ainsi que le voisinage $S3R$ améliore seulement de 1.92% alors que le voisinage $S2R$, qui est beaucoup plus petit, améliore de 3.14%. Le voisinage $S3T$ est 40% meilleur que le voisinage $S2T$, ce qui est beaucoup plus important que pour le TTP. Cela est dû au nombre d'équipes plus élevé dans le BSSP. Finalement, on remarque que le voisinage SH réussit très mal à améliorer les solutions avec 0.23%.

5.2 VND sans le grand voisinage

Nous avons vu à la section précédente la capacité de chacun des voisinages à optimiser les problèmes dans une descente de recherche locale. Certains se sont révélés plus efficaces que d'autres. Cet exercice avait pour but de mieux caractériser les voisinages pour la conception de la recherche à voisinage variable. Nous présentons dans cette section, les résultats des différentes stratégies de recherche à voisinage variable. Les tableaux 5.14, 5.15, 5.16 et 5.17 sont les résultats pour les différentes

Tableau 5.14: VND pour 10 équipes (moyenne de 100 exécutions)

Type	Valeur solution				Temps			Iter.
	initiale	best	moy	pire	best	moy	pire	
VNDA1	87149.5	67147	72452.5	76937	2.9	6.2	13.4	39.4
VNDA2	87149.5	67774	72712.0	77442	2.4	5.2	11.6	42.0
VNDB1	87149.5	68514	72511.0	77416	2.8	7.7	21.0	39.0
VNDB2	87149.5	68936	72466.1	77442	2.4	7.3	19.4	39.6
VNDC1	87149.5	68243	72192.6	77416	2.9	9.8	18.7	38.2
VNDC2	87149.5	68359	72378.1	77442	2.5	9.1	22.4	36.7
VNDD	87149.5	68289	73081.4	79413	2.4	5.0	10.2	23.4
VNDE	87149.5	68883	72934.2	79413	2.5	6.9	17.2	22.9

Tableau 5.15: VND pour 12 équipes (moyenne de 100 exécutions)

Type	Valeur solution				Temps			Iter.
	initiale	best	moy	pire	best	moy	pire	
VNDA1	162074.4	129583	135753.9	143149	10.8	21.5	48.0	53.0
VNDA2	162074.4	130058	136007.4	141983	8.5	18.2	45.0	58.3
VNDB1	162074.4	128890	135779.2	143149	10.3	25.7	59.2	50.2
VNDB2	162074.4	129761	135167.1	142173	8.7	28.1	82.0	53.2
VNDC1	162074.4	128750	134783.5	143149	11.3	38.3	74.1	50.9
VNDC2	162074.4	129224	134950.7	140967	9.9	37.0	78.9	51.0
VNDD	162074.4	130649	136508.5	142680	8.3	18.6	38.7	32.6
VNDE	162074.4	129347	136446.2	142424	8.3	29.2	93.9	31.7

tailles du TTP et les résultats du BSSP se retrouvent dans le tableau 5.18. La première colonne représente la stratégie de descente à voisinage variable. Les quatre colonnes qui suivent sont les valeurs de : la moyenne des solutions initiales, la meilleure solution trouvée, la moyennes des solutions trouvées et la pire solution trouvée. Les trois colonnes qui suivent sont les temps d'exécution : le temps le plus rapide, la moyenne des temps et le temps le plus lent. Finalement, la dernière colonne est le nombre d'itérations.

En étudiant tous les résultats du TTP et du BSSP, nous voyons que la stratégie

Tableau 5.16: VND pour 14 équipes (moyenne de 100 exécutions)

	Valeur solution				Temps			
Type	initiale	best	moy	pire	best	moy	pire	Iter.
VNDA1	305741.7	233143	245962.5	263779	32.5	61.7	99.5	81.2
VNDA2	305741.7	233282	247377.5	262782	21.4	44.9	98.0	84.2
VNDB1	305741.7	233567	246144.6	261983	30.4	81.2	194.1	77.3
VNDB2	305741.7	232549	246961.1	259556	21.4	76.5	179.9	74.0
VNDC1	305741.7	229377	244177.4	261983	55.9	127.8	228.9	74.9
VNDC2	305741.7	234729	244961.4	259143	21.6	110.0	235.0	72.8
VNDD	305741.7	235180	247774.5	262173	29.0	56.1	107.3	50.7
VNDE	305741.7	234386	246372.5	257190	29.0	107.0	263.6	50.7

Tableau 5.17: VND pour 16 équipes (moyenne de 36 exécutions)

	Valeur solution				Temps			
Type	initiale	best	moy	pire	best	moy	pire	Iter.
VNDA1	432762.2	331368	345650.9	360620	81.6	128.0	231.0	92.6
VNDA2	432762.2	335479	345102.2	355960	46.3	83.3	138.0	96.3
VNDB1	432762.2	324967	343432.1	356630	89.5	204.6	433.3	93.4
VNDB2	432762.2	329478	342824.8	360055	60.2	162.5	373.8	87.5
VNDC1	432762.2	326074	339613.0	355625	160.8	322.6	509.8	88.4
VNDC2	432762.2	326520	340747.9	353917	125.3	277.7	480.1	86.8
VNDD	432762.2	330796	346176.8	357941	71.6	121.8	186.0	63.1
VNDE	432762.2	327569	343296.3	357941	72.5	256.1	770.3	65.8

Tableau 5.18: VND pour 24 équipes (moyenne de 25 exécutions)

Type	Valeur solution				Temps			Iter.
	initiale	best	moy	pire	best	moy	pire	
VNDA1	911221.5	815341	835525.3	862745	363.9	765.8	1556.2	94.5
VNDA2	911221.5	814540	835067.2	866096	308.4	894.9	1862.7	70.2
VNDB1	910965.0	799389	831417.4	864848	309.0	1754.9	3631.9	76.8
VNDB2	911221.5	810214	832979.2	867817	312.7	1617.7	4643.7	71.8
VNDC1	911221.5	799389	830403.4	862407	311.2	1788.8	3434.9	73.8
VNDC2	911221.5	814185	833892.7	856318	401.7	1824.8	4596.9	67.7
VNDD	911221.5	818727	831598.5	869398	538.0	1177.8	2532.9	59.1
VNDE	911034.0	819691	832163.0	869398	652.8	1769.6	3492.9	55.5

VNDC1 est toujours celle qui obtient les meilleurs résultats. La stratégie VNC2 est la seconde meilleure pour le TTP, alors que pour le BSSP il s'agit de la stratégie VNDA1. Pour mieux comprendre ces résultats, nous présentons dans les tableaux 5.19 et 5.20 les nombres d'itérations de chacun des voisinages. Ces deux tableaux nous montrent l'utilisation de chacun des voisinages pendant la recherche. Les stratégies VNDA1, VNDA2, VNDB1 et VNDB2 débutent avec un voisinage et utilisent les autres pour dépanner lorsqu'un optimum local est atteint. Pour les stratégies VNDA1 et VNDB1 on voit par exemple que la recherche débute avec le voisinage $S2R$ qui est utilisé majoritairement, les autres voisinages sont utilisés selon la suite $\langle S2R, S2T, S3R, S3T, NSH \rangle$. On voit aussi que les voisinages NSH et SH sont utilisés en dernier recours; les solutions sont alors beaucoup améliorées ce qui explique le très petit nombre d'itérations. En utilisant principalement les voisinages rapides à explorer, les stratégies VNDA1 et VNDA2 sont beaucoup plus rapides que les autres. Les stratégies VNDC1 et VNDC2 utilisent les voisinages selon une boucle; ainsi, lorsqu'un voisinage est bloqué il cède sa place au suivant. Cette façon de procéder montre que les voisinages sont utilisés plus uniformément. C'est possiblement ce qui permet à cette stratégie d'obtenir les meilleurs résultats. Les stratégies VNDD et VNDE explorent plusieurs voisinages à chaque itération et

Tableau 5.19: Nombre d'itérations par mouvement pour le TTP

Stratégie	Nb	Iter	S2R	S2T	S3R	S3T	NSH
VNDA1	36	92.6	71.9	12.5	4.7	2.6	0.9
VNDA2	36	96.3	20.6	70.9	1.1	3.0	0.6
VNDB1	36	93.4	57.5	17.6	7.4	9.1	1.8
VNDB2	36	87.5	32.4	40.3	3.6	9.9	1.3
VNDC1	36	88.4	41.8	11.8	10.4	16.1	8.3
VNDC2	36	86.8	22.9	33.2	8.0	16.1	6.6
VNDD	36	63.1	59.5	59.5	3.6	3.6	3.6
VNDE	36	65.8	52.4	52.4	13.3	13.3	13.3

Tableau 5.20: Nombre d'itérations par mouvement pour le BSSP

Stratégie	Nb	Iter	S2R	S2T	S3R	S3T	SH
VNDA1	25	94.5	54.7	31.5	3.5	3.4	1.4
VNDA2	25	70.2	6.4	53.1	2.9	4.9	2.9
VNDB1	25	76.8	19.0	36.5	3.9	13.2	4.3
VNDB2	25	71.8	6.2	47.6	2.0	11.8	4.2
VNDC1	25	73.8	15.0	36.2	3.3	13.3	6.0
VNDC2	25	67.7	4.6	42.1	2.8	13.2	5.0
VNDD	25	59.1	53.1	53.1	6.0	6.0	6.0
VNDE	25	55.5	45.2	45.2	10.3	10.3	10.3

gardent le meilleur voisin, c'est pourquoi, par exemple, que les voisinages $S2R$ et $S2T$ ont les mêmes nombres d'itérations. Ces stratégies cherchent donc à aller le plus rapidement possible à la meilleure solution. La recherche est donc plus directe vers un optimum local. Nous choisissons d'utiliser la stratégie VNDC1, car elle obtient en moyenne de meilleures solutions pour le TTP et le BSSP.

5.3 VND avec grand voisinage

Dans cette section, nous présentons les résultats obtenus lorsque nous ajoutons le grand voisinage dans la recherche à voisinage variable. Le grand voisinage consiste à fixer un certain nombre de tournées dans l'horaire et à utiliser le modèle de programmation par contraintes pour chercher de nouvelles solutions. Il y a trois paramètres à déterminer pour le grand voisinage : le nombre de tournées à fixer, le temps accordé pour l'exploration de chaque préaffectation et le temps de recherche total. Pour le TTP, nous avons accordé douze heures pour la recherche complète et le double pour le BSSP. Ces temps sont conservateurs en comparaison avec certains temps de calculs trouvés dans la littérature. Il faut comprendre que l'objectif premier de ce mémoire est de travailler avec le modèle de recherche locale en programmation par contraintes et non pas de battre à tout prix les résultats de la littérature. Nous avons ensuite limité l'exploration des préaffectations à trente secondes pour le TTP et trois cents pour le BSSP. Le nombre m de tournées préaffectées à été choisi pour que l'exploration se fasse en quelques secondes. Ainsi, c'est à la deuxième phase de la recherche, lorsque nous affectons $m-1$ tournées, que la limite est réellement utilisée et cela dans le but de ne pas perdre trop de temps dans les mauvaises préaffectations. Le tableau 5.21 résume les valeurs que nous avons choisies pour les paramètres selon la taille du problème. La stratégie d'exploration pour les préaffectations du grand voisinage consiste à affecter les variables ayant le plus petit domaine d'abord et de choisir les valeurs selon l'ordre lexicographique. Nous avons cherché de meilleures stratégies sans succès.

Le tableau 5.22 présentent les résultats moyens obtenus par la descente à voisinage variable avec et sans le grand voisinage (GV). Pour la VND avec le grand voisinage, nous présentons ici aussi la meilleure solution trouvée, la moyenne des solutions trouvées et la pire solution trouvée. La dernière colonne est le nombre d'itérations

Tableau 5.21: Valeurs des paramètres de la VND

Nombre d'équipes n	10	12	14	16	BSSP
Nombre de préaffectations	4	6	8	10	14
Temps par préaffectation (S)	30	30	30	30	300
Temps total (S)	43200	43200	43200	43200	86400

Tableau 5.22: VND avec grand voisinage (moyenne de 10 exécutions)

n	Solution initiale	VND sans GV	VND avec GV			Temps	Iter.
			Meilleur	Moyen	Pire		
10	88193.6	72991.9	65321	67767.8	72163	8304.9	7.2
12	163266.2	136369.4	124713	127578.9	130429	30923.7	10.4
14	308668.4	248860.9	218547	224222.1	231620	43214.4	18.3
16	433798.1	344323.6	309290	317471.4	324817	43214.5	16.3
BSSP	904072.8	828704.4	800456	809275.4	814844	86404.0	6.2

de grand voisinage, c'est-à-dire le nombre de fois que le grand voisinage a été utilisé. Le tableau 5.23 présente l'amélioration des solutions en terme de pourcentage. On voit que la recherche à voisinage variable sans le grand voisinage améliore les solutions du TTP d'environ 18%. Avec le grand voisinage, l'amélioration passe à environ 25%. Pour le BSSP, l'amélioration sans grand voisinage est de 8.34% et monte à 10,49% avec. La limite de trente secondes aurait pu être plus élevée pour les tailles 10 et 12 du TTP, car on voit que les douze heures n'ont pas été complètement utilisées.

Le tableau 5.24 compare les meilleures solutions obtenues pour le TTP avec celles obtenues par trois autres méthodes. Nos résultats sont meilleurs que ceux de la méthode de relaxation lagrangienne de Benoist, Laburthe et Rottembourg [4] et ils sont comparables à la série de résultats de Cardemil 2002 prise sur le site Internet officiel de ce challenge [8]. Nous sommes cependant loin des meilleures solutions à ce jour pour le TTP trouvées par le recuit simulé d'Anagnostopoulos et al. [2]. La

Tableau 5.23: Amélioration de la descente à voisinage variable

n	VND sans GV %	VND avec GV %	Total %
10	17.24	5.92	23.16
12	16.47	5.38	21.86
14	19.38	7.98	27.36
16	20.63	6.19	26.82
BSSP	8.34	2.15	10.49

Tableau 5.24: Comparaison avec résultats de d'autres méthodes

n	Benoist	Cardemil	Anagnos	VND	Écart %
10	68691	66037	59583	65321	9.63
12	143655	125803	111248	124713	12.10
14	301113	205895	189766	218547	15.17
16	437273	308413	267194	309290	15.75

dernière colonne est l'écart entre ces meilleurs résultats et notre méthode. Nous voyons que cet écart varie autour de 15%, ce qui est très important lorsque l'on considère que la méthode de recuit simulé obtient très rapidement des solutions beaucoup meilleures aux nôtres. Nos voisinages de recherche locale sont pourtant inspirés de cette méthode. Cependant, la méthode de recuit simulé permet l'exploration du domaine non réalisable, alors que nous nous restreignons au domaine réalisable. De plus, cette méthode n'explore pas tout le voisinage mais choisit un voisin au hasard. Nous pensons que c'est ce qui explique la supériorité et la grande efficacité de cette méthode.

Le BSSP est un problème plus récent et moins connu que le TTP. À ce jour, seulement Biajoli et al. [5] se sont attaqués à ce problème. Ils comparent leur résultat à la solution obtenue faite à la main en décembre 2003 par la Brazilian Soccer Confederation (CBF). La solution de Biajoli et al. vaut 842789 alors que

celle de la CBF vaut 991926. Nos meilleures solutions pour le BSSP tournent autour de 800000, soient 800456 obtenu avec le grand voisinage et 799389 obtenu sans même utiliser le grand voisinage. Notre méthode trouve donc des solutions qui sont 19% meilleures que la solution de la CBF et 5% meilleures que la meilleure solution connue. Il est intéressant de rappeler que la méthode de Biajoli et al. est un recuit simulé directement inspiré de la méthode d'Anagnostopoulos et al. pour le TTP.

CONCLUSION

Les objectifs de ce mémoire étaient de tester et de tenter de raffiner le modèle hybride permettant une recherche locale en programmation par contraintes proposé par Pesant et Gendreau [26, 27]. Pour cela, nous avons choisi d'appliquer ce modèle au «Traveling Tournament Problem» et au «Brazilian Soccer Scheduling Problem». Ces deux problèmes ont été choisis car ils sont très contraints et qu'ils sont difficiles à résoudre autant pour les méthodes de recherche opérationnelle que celles de programmation par contraintes. Ils sont donc de bons problèmes pour les méthodes hybrides. Notre application du modèle hybride prend la forme d'une descente à voisinage variable. Ce modèle de recherche locale basée sur les contraintes offre un bonne séparation entre la modélisation du problème et la modélisation de la recherche locale. Cela nous a permis de réutiliser le code écrit pour le TTP et de l'adapter très facilement pour le BSSP. La modélisation du problème a dû être modifiée aux exigences du BSSP, mais la recherche locale n'a nécessité que quelques ajustements. Mettre en oeuvre le BSSP a ainsi été très rapide. Le modèle permet donc d'adapter facilement le code développé à un problème voisin. La communication entre le problème et la recherche locale se fait à l'aide de contraintes d'interface. Ces contraintes traduisent le choix partiel ou complet d'un voisin vers le modèle du problème pour mener à la nouvelle solution partielle ou complète. Ces contraintes permettent ainsi d'éliminer les affectations partielles de voisins menant à des solutions non valides, mais aussi de filtrer les domaines des variables de voisinage qui ne sont pas affectées. L'efficacité de ce modèle hybride dépend beaucoup de ces contraintes. Les contraintes d'interface définissant la nouvelle solution sont faciles à trouver, car elles ne font que traduire le mouvement de recherche locale. Les contraintes permettant de filtrer les variables de voisinages sont cependant plus difficiles à trouver, car elles demandent un algorithme de filtrage.

Ces travaux nous ont menés à modifier la formulation des contraintes d'interface. En effet, les contraintes d'interface du modèle hybride de Pesant et Gendreau [26, 27] sont exprimées en spécifiant dans celles-ci leurs conditions de réveil. Ces conditions de réveil consistent à attendre que les variables de voisinages impliquées soient affectées. Nous avons réalisé qu'il peut être utile d'activer ces contraintes après une simple modification du domaine d'une des variables impliquées. Nous proposons ainsi une nouvelle formulation des contraintes d'interface sans les conditions de réveil. Cette nouvelle forme de contrainte a pour but d'utiliser plus activement les contraintes en incitant la recherche d'algorithme de filtrage pour des modifications plus élémentaires des domaines. Ainsi il est possible d'élaborer des traitements demandant de réveiller les contraintes simplement lorsqu'une valeur est éliminée du domaine ou lorsque les bornes sont resserrées. Bien sûr, il est toujours possible d'attendre que les variables soient affectées comme dans l'ancienne formulation. Nous avons donc exprimé les contraintes d'interface dans ce mémoire sans conditions de réveil. Cette nouvelle formulation nous a poussé à créer un traitement qui, pour certaines contraintes, peut être effectué à une modification aux bornes des domaines des variables impliquées. Trouver de tels traitement est plus difficile que de simplement traduire le mouvement de recherche locale, car les algorithmes de filtrage oeuvrant sur les modifications élémentaires des domaines ne sont pas toujours facile à concevoir. Pour cette raison et parce que nous avons élaboré cette nouvelle formulation tard dans ce projet, il n'y a qu'un petit nombre de contraintes d'interface pour lesquelles nous avons un tel traitement. Les autres contraintes ont un traitement qui est activé lorsque les variables impliquées sont affectées. Nous avons testé l'impact des traitements résultants de la nouvelle formulation dans certains voisinages de la recherche à voisinage variable.

La descente à voisinage variable que nous proposons est constituée de voisinages de recherche locale et d'un grand voisinage. Pour le TTP, nous avons réutilisé les

voisinages de recherche locale *PartialSwapTwoRounds* et *PartialSwapTwoTeams* de la littérature et proposé les nouveaux voisinages *PartialSwapThreeRounds*, *PartialSwapThreeTeams* et *NewSwapHomes*. Nous n'avons pas pu réutiliser le voisinage *NewSwapHomes* pour le BSSP, sa structure miroir nous a forcé à remplacer ce voisinage par le voisinage *SwapHomes* aussi trouvé dans la littérature. Les quatre autres voisinages ont pu être réutilisés pour le BSSP dans leur même forme que pour le TTP. Ces voisinages ont été conçus selon le modèle hybride de recherche locale basée sur les contraintes. Lorsque tous ces voisinages sont bloqués dans des optimums locaux, nous proposons un voisinage différent. Il s'agit d'un grand voisinage conçu selon la technique de recherche à grand voisinage (LNS). Le grand voisinage que nous proposons consiste à préaffecter l'horaire de m équipes parmi n et toutes les combinaisons d'affectations sont explorées jusqu'à l'obtention d'une meilleure solution. Il sert lorsque les voisinages de recherche locale sont tous bloqués dans un optimum local. Nous avons dû définir les stratégies d'exploration de chacun des voisinages ainsi que la stratégie pour la descente à voisinage variable

Pour les voisinages d'échanges entre les rondes, nous avons découvert des classes de mouvements équivalents pour les variables représentant les équipes. Nous avons montré que les meilleures stratégies de base pour l'exploration de ces voisinages consistent à affecter en premier les variables de rondes afin de pouvoir profiter de ces classes d'équivalence. Pour les voisinages d'échanges entre les équipes, nous avons découvert des classes de mouvements équivalents pour les variables de rondes. Dans ce cas, les meilleures stratégies de base consistent à affecter d'abord les variables d'équipes pour pouvoir profiter des classes d'équivalence. Nous avons aussi montré que la technique de fragmentation de domaine permet d'explorer plus rapidement les voisinages. Pour les voisinages *PartialSwapTwoRounds* et *PartialSwapThreeRounds*, nous avons proposé un nouveau traitement permettant de réveiller certaines contraintes lors de modifications aux bornes des domaines des

variables impliquées. Nous avons comparé ce nouveau traitement avec un traitement qui attend l'affectation des variables. Les résultats ont montré que le nouveau traitement est plus rapide même si les nombres d'échecs ne changent pas d'un traitement à l'autre. Cela nous est difficile à expliquer, nous pensons que le nouveau traitement est plus rapide à exécuter pour le solveur. Pour la stratégie du plus petit domaine d'abord sans la fragmentation de domaine, les temps d'exécution sont plus lents avec le nouveau traitement. Nous pensons que cela est causé par le fait que, comparativement aux autres stratégies, le nouveau traitement appelle beaucoup plus souvent la fonction d'évaluation des combinaisons pendant l'exploration. Nous considérons que les meilleures stratégies sont celles qui sont les plus rapides pour nos problèmes. Certaines stratégies peuvent réussir à élaguer une très grande portion de l'arbre de recherche et ainsi en explorer qu'un faible pourcentage, si pour en arriver à cela il faut cependant des algorithmes très lents les bénéfices de l'élagage sont perdus. Il faut par contre considérer que ces algorithmes peuvent être utiles pour des arbres de recherche plus grands où les effets de l'élagage seraient plus prononcés. Il faut donc trouver le compromis le plus efficace.

La descente à voisinage variable a permis d'optimiser les solutions de départ trouvées avec les modèles de programmation par contraintes des problèmes. La descente sans le grand voisinage améliore les solutions du TTP d'environ 18% et environ 25% avec le grand voisinage. Pour le BSSP, l'amélioration sans grand voisinage est de 8.34% et monte à 10.49% avec celui-ci. La technique de recherche à grand voisinage est une bonne méthode pour dépanner les voisinages de recherche locale lorsque ceux-ci sont bloqués dans des optima locaux. Notre grand voisinage ne semble cependant pas très efficace. Une des principales raisons est que nous n'avons pas trouvé une bonne stratégie d'exploration pour les préaffectations. Nous avons donc dû considérer des préaffectations pouvant être explorées rapidement afin de ne pas perdre tout le temps dans les mauvaises préaffectations. Pour que la technique de

grand voisinage soit plus efficace, il faut assurément trouver une meilleure stratégie d'exploration. Cela permettrait de préaffecter l'horaire de moins d'équipes et offrirait ainsi un plus grand voisinage. Nous avons exploré sans grand succès plusieurs pistes pour trouver un meilleur grand voisinage pour le TTP. Nous pensons tout de même qu'il est possible de trouver un voisinage efficace où il serait possible d'identifier les bonnes décisions de la solution courante.

Finalement, nous tenons à mentionner que la nouvelle forme des contraintes d'interface est un ajout important au modèle de Pesant et Gendreau [26, 27], car elle permettra d'utiliser plus activement ces contraintes. Il serait donc intéressant de poursuivre les travaux dans cette direction.

BIBLIOGRAPHIE

[1] F. Ajili et H. El Sakkout : LP probing for piecewise linear optimization in scheduling. *Proceedings CPAIOR 2001*, pages 189–203, 2001, Ashford, Angleterre.

[2] A. Anagnostopoulos, L. Michel, P. Van Henteryck et Y. Vergados : A Simulated Annealing Approach to the Traveling Tournament Problem. *Proceedings CPAIOR 2003*, 2003, Montréal, Canada.

[3] D. Applegate et W. Cook : A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal On Computing*, 3:149–156, 1991.

[4] T Benoist, F Laburthe et B Rottembourg : Lagrange Relaxation and Contraint Programming Collaborative schemes for Traveling Tournament Problems. *Proceedings CPAIOR 2001*, 2001, Ashford, Angleterre.

[5] F. J. Biajoli, M. J. Souza, A. A. Chaves et O. M. Mine : Scheduling the Brazilian Soccer Championship: A Simulated Annealing Approach. *Practice and Theory of Automated Timetabling 2004*, 2004, Pittsburgh, États-Unis.

[6] S Bourdais, P. Galinier et G. Pesant : HIBISCUS: a constraint programming application to staff scheduling in health care. *Proceedings of the International Conference on Constraint Programming - CP 2003*, Lecture Notes in Computer Science 364:153–167, 2003, Kinsale, Irlande.

[7] S. Chamberland et S. Pierre : On the Design Problem of Cellular Wireless Networks. *publication Centre de recherche sur les transports*, CRT-2002-38, 2002.

[8] K. Easton, G. Nemhauser et M. Trick : The Traveling Tournament Problem Description and Benchmarks. *Proceedings of the International Conference*

on Constraint Programming - CP 2001, Springer Lecture Notes in Computer Science 2239:580–585, 2001, Paphos, Chypre.

[9] K. Easton, G. Nemhauser et M. Trick : Solving the Traveling Tournament Problem: A Combined Integer Programming and Constraint Programming Approach. *Practice and Theory of Automated Timetabling IV*, Springer Lecture Notes in Computer Science 2740:100–109, 2002, Gent, Belgique.

[10] F. Focacci, F. Laburthe et A. Lodi : chapitre Local Search and Constraint Programming. *dans Handbook of Metaheuristics*, F. Glover et G. Kochenberger (Éditeurs), Kluwer Academic Publishers, 2003.

[11] F. Glover : Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research*, 1:190–206, 1986.

[12] F. Glover : Tabu Search - Part I. *ORSA Journal on Computing*, 2:4–32, 1989.

[13] F. Glover : Tabu Search - Part II. *ORSA Journal on Computing*, 2:4–32, 1989.

[14] P. Hansen et N.F Mladenovic : chapitre An introduction to variable neighborhood search. *dans Meta-heuristics, Advances and Trends in Local Search Paradigms for Optimization*, S. Voss, S. Martello, I. H. Osman et C. Roucairol (Éditeurs), Kluwer Academic Publishers, 1998.

[15] W. D. Harvey et M. L. Ginsberg : Limited discrepancy search. *Proceedings of the 14th International Joint Conference on Artificial Intelligence - IJCAI 1995*, Morgan Kaufmann:607–615, 1995, Édinbourg, Écosse.

[16] W.J. van Hoeve : The Alldifferent Constraint: A Survey. *In Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001, Prague, République Tchèque. http://www.arxiv.org/html/cs/0110012.

[17] ILOG : ILOG Solver 6.0 Reference Manual. 2003.

[18] ILOG : ILOG Solver 6.0 User's Manual. 2003.

[19] N. Jussien et O. Lhomme : Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139:21–45, 2002.

[20] O. Kamarainen et H. El Sakkout : Local Probing Applied to Scheduling. *Proceedings of the International Conference on Constraint Programming - CP 2002*, Springer Lecture Notes in Computer Science 2470:155–171, 2002, Ithaca, États-Unis.

[21] S. Kirkpatrick, C. D. Gelatt Jr et M. P. Vecchi : Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[22] F. Laburthe et Y. Caseau : SALSA : A Language for Search Algorithms. *Proceedings of the International Conference on Constraint Programming - CP 1998*, Springer Lecture Notes in Computer Science 1520:310 – 324, 1998, Pise, Italie.

[23] L. Michel et P. Van Hentenryck : Localizer: A Modeling Language for Local Search. *Proceedings of the International Conference on Constraint Programming - CP 1997*, Springer Lecture Notes in Computer Science 1330:237–251, 1997, Schloss Hagenberg, Autriche.

[24] N.F Mladenovic et P. Hansen : Variable Neighborhod Search. *Computers and Operations Research*, 24:1097–1100, 1997.

[25] G. Pesant : A Filtering algorithm for the stretch constraint. *Proceedings of the International Conference on Constraint Programming - CP 2001*, Lecture Notes in Computer Science 2239:183–195, 2001, Paphos, Chypre.

[26] G. Pesant et M. Gendreau : A view of local search in constraint programming. *Proceedings of the International Conference on Constraint Programming - CP 1996*, Lecture Notes in Computer Science 1118:353–366, 1996, Massachusetts, États-Unis.

[27] G. Pesant et M. Gendreau : A Constraint Programming Framework for Local Search Methods. *Journal of Heuristics*, 5:255–279, 1999.

[28] L.M. Rousseau, M. Gendreau et G. Pesant : Using Constraint-Based Operators to Solve the Vehicle Routing Problem with Time Windows. *Journal of Heuristics*, 8:43–58, 2002.

[29] P. Shaw : Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Proceedings of the International Conference on Constraint Programming - CP 1998*, Springer Lecture Notes in Computer Science 1520:415–431, 1998, Pise, Italie.

[30] P. Shaw, B. De Backer et V. Furnon : Improved Local Search for CP Toolkits. *Annals of Operations Research*, 115:31–50, 2002.

[31] P. Van Hentenryck et L. Michel : Control Abstractions for Local Search. *Proceedings of the International Conference on Constraint Programming - CP 2003*, Springer Lecture Notes in Computer Science 2833:65–81, 2003, Kinsale, Irlande.

[32] T. Walsh : Depth-bounded discrepancy search. *Proceedings of the 15th International Joint Conference on Artificial Intelligence - IJCAI 1997*, Morgan Kaufmann:1388 –1393, 1997, Hyderabad, Inde.